2-2005

Reconfigurable elliptic curve cryptography

Aarti Malik

Reconfigurable Elliptic Curve Cryptography

by

Aarti Malik

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Dr. Pratapa Reddy, Professor
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY
February 2005

Approved By:

Pratapa Reddy

Dr. Pratapa Reddy, Professor
Primary Advisor – R.I.T. Dept. of Computer Engineering

Greg P. Semeraro

Dr. Greg Semeraro, Assistant Professor
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Dorin Patru

Dr. Dorin Patru, Assistant Professor
Secondary Advisor – R.I.T. Dept. of Electrical Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title: Reconfigurable Elliptic Curve Cryptography

I, Aarti Malik, hereby grant permissions to Wallace Memorial Library to reproduce my thesis in whole or part.

Aarti Malik

Aarti Malik

02 / 07 / 05

Date

Acknowledgements

I would like to thank all my committee members, Dr. Reddy, Dr. Semeraro and Dr. Patru for all their time, guidance and support in helping me in the completion of my thesis.

Abstract

Reconfigurable Elliptic Curve Cryptosystem

Aarti Malik

Supervising Professor: Dr. P. Reddy

Elliptic Curve Cryptosystems (ECC) have been proposed as an alternative to other established public key cryptosystems such as RSA (Rivest Shamir Adleman). ECC provide more security per bit than other known public key schemes based on the discrete logarithm problem. Smaller key sizes result in faster computations, lower power consumption and memory and bandwidth savings, thus making ECC a fast, flexible and cost-effective solution for providing security in constrained environments. Implementing ECC on reconfigurable platform combines the speed, security and concurrency of hardware along with the flexibility of the software approach.

This work proposes a generic architecture for elliptic curve cryptosystem on a Field Programmable Gate Array (FPGA) that performs an elliptic curve scalar multiplication in 1.16milliseconds for GF (2^{163}), which is considerably faster than most other documented implementations. One of the benefits of the proposed processor architecture is that it is easily reprogrammable to use different algorithms and is adaptable to any field order. Also through reconfiguration the arithmetic unit can be optimized for different area/speed requirements. The mathematics involved uses binary extension field of the form GF (2^n) as the underlying field and polynomial basis for the representation of the elements in the field. A significant gain in performance is obtained by using projective coordinates for the points on the curve during the computation process.

Contents

Acknowledgments iii

Abstract iv

List of Tables ix

List of Figures x

Glossary xi

1. Introduction 1
   - 1.1 Introduction 1
   - 1.2 Motivation 2
   - 1.3 Thesis Outline 4

2. Past Implementations 6
   - 2.1 Software Implementations 6
   - 2.2 Hardware Implementations 9

3. Cryptography 13
   - 3.1 Introduction 13
   - 3.2 Private Key Cryptosystem 14
   - 3.3 Public Key Cryptosystem 15
   - 3.4 Digital Signatures 17
   - 3.5 Discrete Logarithm Problem 19
   - 3.6 Key Management 20
     - 3.6.1 Diffie Hellman Key Exchange 21
   - 3.7 Certificates 21

4. Mathematical Background 23

4.1 Basic Structures of Algebra 24
4.1.1 Identity Element 24
4.1.2 Groups 24
4.1.3 Rings 26
4.1.4 Fields 26
4.1.5 Some Definitions 27
4.2 Finite Fields 28
4.2.1 Prime Field 30
4.2.2 Binary Field 30
4.3 Polynomial Basis 30
4.4 Optimal Normal Basis 32
4.5 Implementation Options 34

5. Elliptic Curves 37
5.1 Elliptic Curves over Real Numbers 37
5.2 Elliptic Curves over Finite Fields 41
5.2.1 Prime Fields 41
5.2.2 Binary Extension Fields 43
5.3 Elliptic Curve Scalar Multiplication 44
5.4 Elliptic Curve Discrete Logarithm Problem 46
5.5 Key Exchange Protocols for Elliptic Curve Cryptography 47
5.5.1 Elliptic Curve Diffie-Hellman Protocol 47
5.5.2 Elliptic Curve ElGamal Protocol 48
5.6 Elliptic Curve Digital Signature Algorithm 48
5.7 Embedding Raw Data on the Curve 50

6. Algorithms 51
6.1 Irreducible Polynomial 51
6.2 Coordinate representation 52
6.2.1 Projective Space 52
6.2.2 Curve operation in projective coordinates 53

6.3 Point Multiplication Algorithms 54
6.4 Montgomery Point Multiplication Algorithm for GF (2m) 56
6.5 Inversion Algorithm 59
6.5.1 Fermat's Little Theorem 60

7. Design and Implementation 62
7.1 Methodology 62
7.2 Overview of VHDL 63
7.3 CAD Tools 63
7.4 Target FPGA 64
7.5 Platform Options 65
7.5.1 Reconfigurable Hardware 66
7.6 Introduction to FPGA 67
7.6.1 Configurable Logic Blocks or Logic Array Blocks 68
7.6.2 Input/Output Blocks 70

8. Elliptic Curve Processor 71
8.1 Basic Operation 72
8.2 Architecture of Processor 73
8.2.1 Datapath 73
8.2.2 Instruction Set 74
8.2.3 Control Unit 75
8.2.4 Exception Units 75
8.2.5 Register File 76
8.3 Arithmetic Unit Architecture 76
8.3.1 Multiplier 77
8.3.2 Squarer 80
8.3.3 Inversion 81

9. Results 83

9.1 Timing and Area Analysis 83
9.1.1 Addition 83
9.1.2 Multiplication 84
9.1.3 Inversion 86
9.1.4 Squaring 87
9.1.5 Scalar Multiplication 88
9.1.6 Power Consumption 88
9.2 Comparison with other implementations 89

10 Conclusion 90
10.1 Summary 90
10.2 Recommendation for further work 90

Bibliography 92
A Matlab Code for scalar multiplication 100
B Instructions for Scalar Multiplication in Instruction Memory 102

List of Tables

2.1 Summary of Past Implementations of ECC in software 8

2.2 Summary of Past Implementations of ECC on FPGA 11

5.1 Computational Effort for Cryptanalysis of DLP 46

6.1 Irreducible Polynomial bit complexity for arithmetic operations 52

6.2 Number of operations in affine and projective coordinates 53

6.3 Complexity of GF (2^m) arithmetic operations 54

6.4 Comparison of Point Multiplication Algorithms 56

6.5 Computational complexity of Montgomery Point multiplication Algorithm 59

7.1 Features of Implementation Platforms 66

8.1 Format of Instructions 74

8.2 Instruction Set 75

8.3 Time-area analysis for multiplier architectures 77

9.1a Addition results in Flex10K 84

9.1b Addition results in XCV1000 84

9.2a Multiplication results in Flex10K 85

9.2.1b Multiplication results in XCV1000 with digit size 4 85

9.2.2b Multiplication results in XCV1000 with digit size 16 85

9.2.3b Multiplication results in XCV1000 with digit size 32 85

9.3a Inversion results in Flex10K 86

9.3b Inversion results in XCV1000 86

9.4a Squaring results in Flex10K 87

9.4b Squaring results in XCV1000 87

9.5a Scalar Multiplication results in Flex10K 88

9.5b Scalar Multiplication results in XCV1000 88

9.6 Comparison with Past Implementations 89

List of Figures

3.1 Language of Cryptography 13
3.2 Symmetric Cryptosystem 14
3.3 Asymmetric Cryptosystem 15
3.4 Generation of a Digital Signature 18
3.5 Verification of a Digital Signature 18
4.1 Finite Field Options 29
5.1 Elliptic Curves over Real Numbers 38
5.2 Point Addition 38
5.3 Adding point P and –P 39
5.4 Point Doubling 40
5.5 Doubling a point on X-axis 40
5.6 Scalar Multiplication Hierarchy 45
7.1 Platform Options 65
7.2 Basic Xilinx FPGA Structure 67
7.3 Flex 10K Logic Array Block 68
7.4 Block Diagram of Virtex CLB 69
7.5 Xilinx IO Block 70
8.1 High level view of ECP 71
8.2 Basic Processor Operation 72
8.3 Block diagram of processor showing pipeline stages 73
8.4 Arithmetic Unit 76
8.5 Example of Squarer 81

Glossary

A

ANSI American National Standards Institute

ARM Advanced RISC Machine

ASIC Application-Specific Integrated Circuits

C

CAD Computer-Aided Design

CLB Configurable Logic Block

D

DES Data Encryption Standard

DLP Discrete Logarithm Problem

DSA Digital Signature Algorithm

E

ECC Elliptic Curve Cryptography

ECP Elliptic Curve Processor

F

FPGA Field Programmable Gate Array

G

GF Galois Field

L

LUT Look-up Table

LE Logic Element

M

MAC Message Authentication Code

MIPS Microprocessor without Interlocked Pipeline Stages

N

NIST	**National Institute of Standards in Technology**

O

ONB	**Optimal Normal Basis**

P

PCI	**Peripheral Component Interconnect**

R

RISC	**Reduced Instruction Set Computer**

S

SRAM	**Static Random Access Memory**

V

VLSI	**Very Large Scale Integration**

Chapter 1

Reconfigurable Elliptic Curve Cryptosystems

1.1 Introduction

This thesis involves implementing an elliptic curve cryptosystem on an FPGA (Field Programmable Gate Array). The design and algorithm are proposed to provide faster security mechanism than any other known public key scheme, and is targeted for use in constrained environments like mobile devices and smart cards.

Data communication in an open environment poses a threat to its users by compromising the security of every transaction. It is very important to maintain the security, integrity and validity of such transactions. Various cryptosystems have been established to ensure information security. RSA (Rivest Shamir Adleman) is one the most popular public key cryptosystem in use today [PIE00] and is based on the integer factorization problem. Elliptic curve cryptosystem (ECC) was proposed as an alternative to RSA and other public key schemes like Elgamal encryption and Digital Signature Algorithm (DSA), based on the integer factorization or the discrete logarithm problem.

Elliptic Curve Cryptosystems have been standardized by NIST (National Institute of Standards and Technology) [NIST94] for use as a public key scheme. The US government has also adopted ECC for the elliptic curve DSA [USG00]. This cryptosystem is built of complex mathematical operations that are implemented over finite group of numbers. Elliptic curve groups over finite fields (finite field is a group of elements satisfying certain mathematical properties) are very efficient for cryptographic implementations [SIM92]. Also the representation of elements in the finite field, which in this case are the points on the elliptic curve, plays a very important part in determining the ease of implementation [CERT00].

The strength of ECC is based on the intractability of the elliptic curve discrete logarithm problem, which is composed of the group operation of elliptic curves i.e., the elliptic curve scalar or point multiplication. There is no deterministic time algorithm to solve the

elliptic curve discrete logarithm [MEN95]. Most algorithms take exponential time to compute the discrete logarithm, using many supercomputers, and the data is generally obsolete by then [MEN95].

An architecture for an elliptic curve processor is proposed in this work, which is capable of performing the elliptic curve point multiplication operation in the fields GF (2^m). The architecture of the processor is versatile and can adapt to any field order depending on the level of security required (higher field orders tend to offer more security). It is a low-power processor suited for devices with limited energy resources. Another feature is that all the operations for point multiplication are implemented on the chip resulting in lower input-output bandwidth requirements, than some other implementations which include only the finite field arithmetic operations on the chip. Also this design is well suited for implementation on reconfigurable platform.

The functional unit, capable of performing the finite field arithmetic operations, is the most critical unit of the elliptic curve processor. Efficient architectures for the various arithmetic units have been proposed in this work. These include a digit-serial multiplier and a bit-parallel squarer. Also, a significant gain in performance is achieved by using projective coordinates for representation of points on the elliptic curve.

1.2 Motivation

The popularity of the Internet has grown considerably over the past few years. It is very important to maintain security of transactions over such an open environment. This has fueled research into various cryptographic protocols. Also since the Internet provides a diverse environment of heterogeneous systems, it is impossible to find one algorithm that meets the needs of all users. In order to meet these needs, reconfigurable cryptographic implementations can be used, which enable users to select among different algorithms, depending on their particular application.

According to NIST (National Institute of Technology and Standards), elliptic curve cryptosystems (ECC) can be used to provide more security per bit than any other known

public key scheme. They have matched the key lengths of various public-key algorithms with the strength of resulting cryptosystems, for example, a 163-bit key implementation of elliptic curve technology offers equivalent security strength of a 1024-bit RSA system [LD00]. Smaller key sizes have the potential to result in faster mathematical computations (public key schemes involve complex mathematical operations using the key), lower processing power, memory and bandwidth saving. This makes ECC really attractive for mobile devices having limited power, CPU (Central Processing Unit) and bandwidth. Additionally, ECC is not a single algorithm but a family of various public-key algorithms. Therefore the user can choose from the different algorithms for tradeoff between speed and security of the cipher.

One way of providing faster security mechanisms is to embed the security mechanism in hardware. Reconfigurable hardware devices or FPGAs provide more flexibility than traditional ASIC (Application Specific Integrated Circuits) systems for designing cryptographic applications. The flexibility arises from the capability of reconfigurable devices to use different algorithms optimized for specific applications. It could be argued here that a static algorithm operating in different processing modes can also be used to meet the need for different algorithms. However, implementing different processing modes on a single device takes up more resources and increases cost. Reconfiguration helps reduce system costs through hardware reuse [SHA97] by swapping between algorithms as required. Also problems with the design can be fixed without altering any on board resources. Another benefit of reconfigurable hardware is that it can meet the varying security requirements when the mobile device roams in various trust domains. (A trust domain is a security domain which is under the control of a single trusted authority. All the entities in a particular domain trust this authority. Different trust domains have different security policies which the mobile device should adapt to.)

FPGAs also provide the ability to develop faster, area efficient and cost effective designs than software implementations. The instructions in most software languages are executed sequentially and have a low degree of parallelism [DW99] than hardware

implementations. Also, if instruction level parallelism (ILP) techniques such as superscalar or VLIW (Very Long Instruction Word) are used (where the ILP compilers extract the parallel instruction stream from an application and distribute it to different execution units), even the smartest ILP compiler cannot provide a high degree of parallelism by keeping all the execution units running at all times [MAR01]. The processors are not dedicated for a specific purpose, so they are relatively slow and consume more power [DW99]. Operating systems are also incapable of securely maintaining the private key as in hardware [DMV01]. This has motivated the study of elliptic curve cryptosystems on hardware platforms, particularly reconfigurable hardware.

1.3 Thesis Outline

Chapter 2 summarizes the past implementations of elliptic curve systems in both hardware and software platforms. It addresses the issues involved in choosing a particular platform and compares some of the past research contributions. Chapter 3 provides an introduction to cryptography, comparison of public key and private key cryptosystems, and discusses the various cryptographic protocols. In chapter 4, the mathematical background needed for implementing finite field arithmetic is discussed. With the knowledge of the mathematics, elliptic curves are introduced in chapter 5 and the elliptic curve operations needed for the elliptic curve scalar multiplication process are explained. A description of the various elliptic curve cryptographic protocols is also provided in this chapter. The algorithms used for the implementation of the elliptic curve cryptosystem are described in chapter 6 along with their benefits and comparisons with other algorithms. The entire design and implementation cycle is contained in chapter 7. Also the tools used for implementation and their features are discussed. The architecture of the target FPGAs, and their advantages and disadvantages are described. Chapter 8 describes the architecture of the elliptic curve processor. The architecture of the various arithmetic units which perform the finite field arithmetic is also explained, along with its benefits and performance comparisons. Also described is the concurrency used in the design. Chapter 9 contains the results and conclusion. It also contains performance analysis with past implementations. An absolute timing analysis of the system and the individual

functional units, along with recommendations for future research in this area are provided.

Chapter 2

Past Implementations

There have been a number of notable implementations of elliptic curve cryptosystem on both hardware and software platforms. The following summarizes some of the work in this area.

2.1 Software Implementations

A fast implementation of key exchange (exchange of a piece of information used for encryption and decryption) using an elliptic curve was reported in [SOO95]. This was one of the earliest reported implementations of elliptic curve cryptosystem and it compared elliptic curve with non elliptic curve implementations. The elliptic curve group was defined over the binary field GF (2^{155}), using polynomial basis representation for the points on the curve. This software implementation was tested on two platforms, on the SUN SPARC IPC (25 MHz, 32-bit architecture) and the DEC Alpha 3000(175 MHz, 64-bit architecture). They used the double-and-add method for multiplication in the finite field. Squaring was done using a linear time algorithm by interleaving 0s in between the 1s in the input polynomial. A fast and efficient algorithm for computing inverse was also proposed. The scalar multiplication operation was performed in 124 milliseconds on the SUN whereas it took only 9.9 milliseconds on the DEC Alpha; because of the 64-bit architecture of the Alpha the performance was much better. This work also showed the elliptic curve implementations using GF (2^m) were computationally faster than non elliptic curve versions for the same level of security.

In order to further increase the speed of computations, a software implementation over GF (2^m) using polynomial basis representation was described in [DBV96], where the calculations were carried out using pre-computed lookup tables. Since the intermediate values for various operations were already computed and stored, computations did not have to be done each time, which resulted in higher speeds. The field elements were represented as polynomials with coefficients in the smaller field GF (2^{16}), because computations in a smaller field are easier and faster, with less number of bits. This

implementation was tested on a Pentium/133 based PC. A field multiplication operation for GF (2^{177}) took 62.7 microseconds, which was significantly faster than the one reported in the previous case.

Although, binary and prime fields are the two most popular choices for elliptic curve implementations [CERT00], Guajardo and Paar [GP97] investigated an implementation of ECC on composite fields of the form GF ((2^m) ^{n}). They proposed three new and efficient algorithms for elliptic curve arithmetic over composite Galois fields GF ((2^m) ^{n}). These included algorithms for point multiplication, inverse and field multiplication operations. However, a recent attack on ECC over composite fields has made it less useful in practice [GHS00]. They have shown that is relatively easier to solve the discrete logarithm in composite fields.

An implementation of elliptic curve cryptography in C on a Pentium II, 400 MHz processor using binary fields is reported in [DLA00]. The minimum time taken for the point multiplication operation was 1.6milliseconds for GF (2^{163}) using random curves and 1.1 milliseconds using fixed Koblitz curves, which are recommended by NIST (National Institute of Standards in Technology). They used projective coordinates for the representation of the points on the curve which resulted in much better performance than earlier implementations.
Brown et. al. [BDM01] reported a similar implementation over prime fields. Both these implementations used NIST recommended fields and curves.

There was an increased demand for higher- speed encryption algorithms [PT01] which were required to run at the transmission rates of the wireless communication links. Therefore, new methods were proposed to accelerate the finite field arithmetic used in elliptic curve cryptosystems.

Elliptic curve point multiplication involves multiplying a random integer k with a point on the curve P, to get another point on the curve (represented as kP = 2(((P + 2(P + 2P)) + ...+ P))). This is a series of point additions and point doublings. In [SS01] a new method

of computing 2^k P (point doubling operation) directly without computing intermediate points 2P, 2^2P,...2^{k-1}P was introduced which claimed to improve performance by 45% than other known algorithms for point doubling. An elliptic curve scalar multiplication using this method took 18.4 milliseconds on a GF (2^{160}) using Pentium II, 400MHz. Other algorithms for improved arithmetic in finite fields suitable for software platforms were introduced in [KING01, HAS01, OP00a, and LD98].

The following table summarizes the past software implementations. (For implementations on prime field like [BDM01], the elements are integers from 0 to p, represented in binary format in most cases.)

Implementation	**Platform**	**Field and Representation**	**Speed (Point Multiplication)**
[SOO95]	SUN SPARC IPC DEC Alpha 3000	GF (2^{155}) – Polynomial Basis	124 milliseconds 9.9 milliseconds
[DBV96]	Pentium /133	GF (2^{177}) – Polynomial Basis	62.7 microseconds
[GP97]	DEC Alpha 3000	GF $(2^{16})^{11}$ – Polynomial Basis	19.7 milliseconds
[DLA00]	Pentium II, 400 MHz	GF (2^{163}) – Polynomial Basis	1.1 milliseonds
[BDM01]	Pentium II, 400 MHz	GF(192) - bits	681 microseconds

Table 2.1: Summary of Past Implementations of ECC in software

It can be seen from the table, that the architecture and speed of the processor in software implementations, greatly influences the speed of computations. The 64-bit DEC Alpha provides better performance in [S0095] than the 32-bit SUN machine. In [GP97] the implementation was tested on 233 MHz and a 175 MHz DEC Alpha. The former produced much better results than the later. Also it is observed that binary fields with polynomial basis representation are a popular choice, since computations over this field are easier as the coefficients of the polynomials are 0 and 1.

Implementations of ECC in software are flexible and portable. They are medium to low cost solutions developed in a reasonable amount of time. In spite of this, implementations on general-purpose processors do not provide good physical security, as it is difficult to securely store keys on most operating systems (O.S)[DMV01]. This has motivated the research into hardware platforms, which are explored for the implementation of ECC.

2.2 Hardware Implementations

Different hardware platforms were investigated for implementing ECC (custom VLSI, ASIC, FPGA). These have been widely researched due to the benefits of smaller key length and less bandwidth requirements. ECC hardware implementations use less number of transistors, for example a VLSI (Very Large Scale Integration) implementation of 155-bit ECC reported utilized only 11,000 transistors [AMV93] compared with an equivalent strength 512-bit RSA processor, which used 50,000 transistors [IWD92].

An efficient ASIC implementation of Elliptic Curve Cryptosystem was reported in [ST03]. The elliptic curve processor architecture proposed in this work supports both GF (p) and GF (2^n) by using a dual-field multiplier. The processor is also capable of adapting to arbitrary fields, curves and operator sizes. This architecture was synthesized on a 0.13 μm CMOS (Complementary Metal Oxide Semiconductor) standard cell library and the speeds obtained for 160-bit scalar multiplication were 1.44 milliseconds and 0.19 milliseconds for GF (p) and GF (2^n) respectively. The design includes a Montgomery multiplier with an optimized data bus and an on-the-fly redundant binary conversion which greatly enhances performance. This implementation is based on advanced

technology and has superior performance. It is a expensive solution useful for critical security applications.

Reconfigurable hardware has been widely explored for implementing ECC, where the cryptosystem can be reconfigured with different parameters and algorithms depending on the varying security requirements. Some of the notable ones reported include [OP00a, SB03, ROS98, OP01, BDG02, NGC03, and LMW00].

Rosner reported [ROS98] an elliptic curve processor (ECP) developed on an FPGA for composite fields. He reported a speed of 4.47 milliseconds for point multiplication on a curve defined over GF $((2^8)^{21})$. Again, composite fields are considered to be prone to attack and its use is being discouraged in ECC implementations.

An FPGA implementation of a Microcoded Elliptic Curve Cryptographic Processor was reported in [LMW00]. The design was successfully tested on a Xilinx Virtex XCV300-4 for 113 bits; it utilized 1290 slices (An FPGA slice usually consists of two Look-up-Tables (LUT), two flip flops and some carry and control logic) at a maximum frequency of 45 MHz. This implementation performed an elliptic curve scalar multiplication in 3.7 milliseconds. In this implementation, both the field and the curve operations were performed on the chip, and so it did not have high input-output (I/O) bandwidth requirements, as compared to the chips which implement only the field operations. Projective coordinates to represent the points of the curves were used in this work. Using projective coordinates eliminates the need for inversion (which is the most costly finite field operation), in the point addition and point doubling operations.

A comparison between different elliptic curve point multiplications using various hybrid coordinate representations and multiplication algorithms is given in [BDG02]. The study suggests that polynomial basis is the best choice for representation of field elements. Also Montgomery algorithm for multiplication using Jacobian or projective coordinates gives the best performance.

The work proposed in [OP00a] describes an efficient and scalable elliptic curve processor architecture for reconfigurable hardware. This design uses two programmable processors, a bit-parallel squarer, digit serial multiplier and polynomial basis representation of elements in the finite field GF (2^n). The prototypes were implemented on the Xilinx XCV400E-8-BG432 (Virtex E) FPGA and the speed reported for computing an elliptic curve scalar multiplication in this case was 0.21 milliseconds for the field GF (2^{167}). This is the fastest reported implementation of ECC on reconfigurable platform. The same group proposed a scalable architecture for an elliptic curve processor for the curve defined over prime field GF (p) in [OP01]. A special Montgomery multiplier algorithm was implemented, which used precomputed values and therefore this architecture is suitable only for memory-rich hardware. A single point multiplication in this case took approximately 3 milliseconds for an arbitrary point on a curve defined over GF ($2^{192} - 2^{64} - 1$), which was slower than the binary field implementation.

The following table summarizes the implementations of ECC on reconfigurable platform.

Implementation	**FPGA Used**	**Field and Representation**	**Speed (Point Multiplication)**
[ST03]	ASIC - 0.13 μm CMOS	GF(192) GF(2^{160}) – Polynomial Basis	1.44 milliseconds 0.19 milliseconds
[ROS98]	XC4062XLPG475-1	GF ((2^8)^{21}) – Polynomial Basis	4.47 milliseconds
[LMW00]	Xilinx Virtex XCV300-4	GF (2^{113}) Optimal Normal Basis	3.7 milliseconds
[OP00]	XCV400E-8-BG432	GF (2^{167}) Polynomial Basis	0.21 milliseconds
[OP01]	XCV1000E-8-BG680	($2^{192} - 2^{64} - 1$) Bits	3 milliseconds

Table 2.2: Summary of Past Implementations of ECC on FPGA

The speed of a particular implementation depends on various factors such as the parameter selection, selection of algorithms, selection of underlying field and of the elliptic curve [CERT00]. The speed of ECC on reconfigurable hardware also depends on the architecture of the target FPGA [DMV01]. Both [LMW00] and [OP00] use projective coordinates for the computation process which has shown to improve performance. Also, fixed elliptic curves have proven to be more efficient in these implementations.

An implementation of a scalable and efficient elliptic curve processor, capable of performing the scalar multiplication operation is reported in this work. The implementation differs from most previous ones by using random elliptic curves defined over binary fields GF (2^n) with polynomial basis representation of elements. Support for arbitrary fields, parameters is also provided in the processor. Fast and efficient algorithms for scalar multiplication and finite field arithmetic are used and prototyped on an FPGA.

Chapter 3

Cryptography

3.1 Introduction

Cryptography is defined as the science of mathematics used to encrypt and decrypt data. Cryptanalysis is the study of how to compromise cryptographic measures, which also aids in designing good ciphers, and cryptology is the combination of cryptography and cryptanalysis. Encryption of data is to transform it into an unintelligible form that can be transmitted or stored securely over insecure systems such as the Internet. Decryption of data is transforming the encrypted data back to readable form.

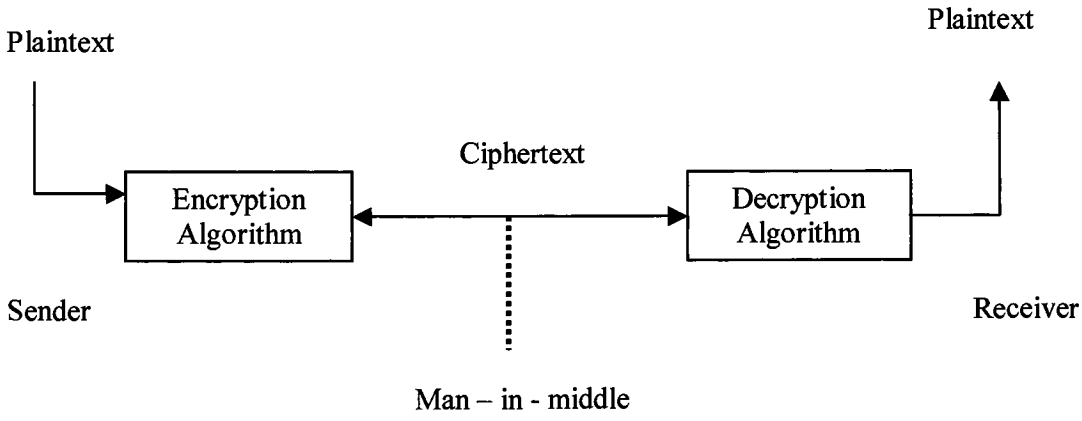


Fig 3.1: Language of Cryptography (The basic cryptographic process)

The figure above illustrates the basis of a cryptographic process. Plaintext is the data that needs to be encrypted. This is done by using some scheme or algorithm that is known as the cipher. The resulting data after application of the cipher is the ciphertext. The cipher-text should be such that any adversary in the communication channel should not be able to decrypt the data back to its original form. Only the person for who the message is

intended can decrypt it using a key. A key is a short piece of information which is used along with a cipher for encryption and decryption, only by an authorized user. An algorithm (cipher) and key together form a cryptosystem.

However, cryptography is not just encryption and decryption. It also provides other services such as authentication, integrity and non-repudiation. Authentication allows the recipient to confirm the identity of the sender. Integrity of the message assures the recipient that it was not modified in transit. It should be noted here that the recipient is only able to detect message modification, not prevent it. Non-repudiation prevents the sender from claiming at a later date that they never sent the message.

There are basically 2 types of cryptosystems that provide the above mentioned services.

1. Public-key or Asymmetric cryptosystem
2. Private-key or Symmetric cryptosystem

3.2 Private-key Cryptosystem

In this type of cryptosystem the key used to encrypt and decrypt is the same, known as the symmetric key.

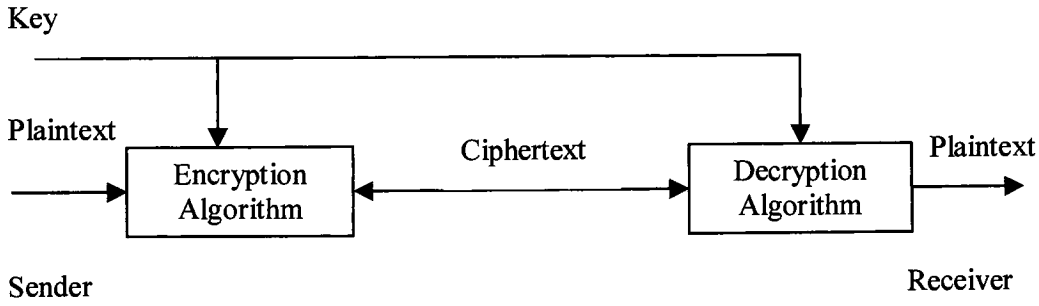


Fig 3.2: Symmetric Cryptosystem – same key is used for encryption and decryption

Symmetric key systems can be used to provide authentication along with encryption. The main problem with this type of cryptosystem is that the different parties involved in the

communication have to agree upon a secret key and keep it secret. There has to be an exchange of key information for the first time. In order to transmit the key it has to be encrypted which requires another key.

The most common techniques used in symmetric key cryptography are block ciphers, stream ciphers and message authentication codes (MACs). Block cipher is an algorithm that encrypts data in discrete units called blocks rather than a continuous stream. A block of plaintext is converted into a block of ciphertext using a user-defined key. The ciphertext is decrypted into plaintext using the same key. A stream cipher encrypts data serially; it works on smaller units of the plaintext. A message authentication code is an authentication tag derived by applying an authentication scheme, together with a secret key to a message. Since MACs are computed and verified using the same key, they can only be verified by the intended recipient.

3.3 Public-Key Cryptosystem

In this type of cryptosystem the key used to encrypt and decrypt is different and hence it is known as asymmetric cryptosystem.

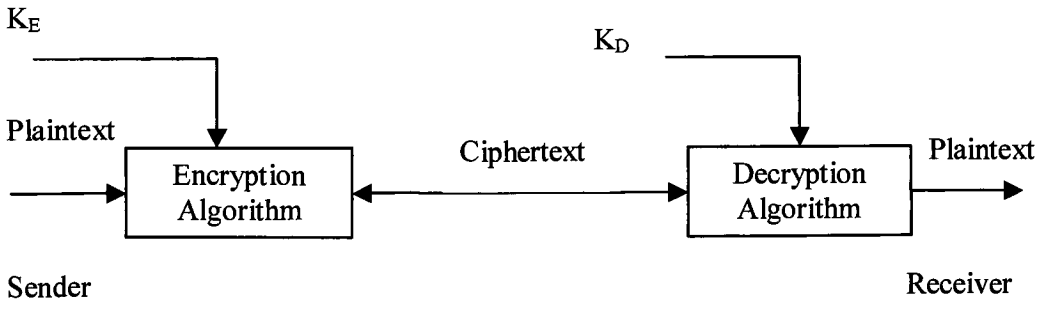


Fig 3.3: Asymmetric Cryptosystem – different key is used for encryption and decryption.

Public-key cryptography was introduced by Whitfield Diffie and Martin Hellman [DH76] to address the key management issue in traditional secret key cryptography. Each person in the communication channel has 2 types of keys – a public key, which is published, and a secret key, which is kept secret. So when A wants to send a message to B, she uses B's public key to encrypt it, and B can decrypt the message using her private key. The two users can communicate securely without the need for exchanging keys. Public key cryptosystems can also be used to provide authentication. If A encrypts a message using his private key then B can decrypt the message using only A's public key, which means that the message was created by A alone.

The public key is used to generate a transformation and the private key is used to generate another transformation. Both transformations are formed of a one-way function and are inversely related to each other. So the public key of a particular user can be used to decrypt a message encrypted by his private key and vice versa. A public key cannot decrypt a message encrypted by another public key.

The main uses of public key cryptography are encryption and authentication using digital signatures.

Advantages and disadvantages of public and private key cryptography

The main advantage of public key cryptosystems is that they do not require any secret key exchange between the users. Secret key cryptography requires the secret key to be transmitted from the sender to the receiver, either manually or through some communication channel. This compromises the secrecy of the secret key and it is possible for someone to know the secret key en route.

Public key cryptosystems also provide techniques for digital signatures which provide the authentication and non-repudiation service. Authentication by secret key involves sharing a secret, sometimes with a third party as well, so a sender can repudiate a previously sent message by claiming that the shared secret was somehow compromised by the third party [RSA00].

The primary advantage of secret key cryptography is that it can provide more efficient and faster encryption than public-key. Public-key algorithms involve complex mathematics and are relatively slower [WAL91].
Also in public key cryptosystems, the message can be sent only to a single user, since it is encrypted using his/her public key. In secret key cryptography, a message encrypted with a secret key can be published and so all the users having the secret key can decrypt it. Another problem in public key cryptography is that a user may impersonate another user since the public keys are published.

In order to achieve advantages from both schemes, public key cryptography should be used for key exchange and authentication and private key for encryption [ROS99]. This is because private key encryption is faster and asymmetric cryptosystems can be used for securely exchanging keys, without the need to share any secret information.

3.4 Digital Signatures

A digital signature of a message is a piece of data dependent on some secret known only to the sender (signer), and additionally, on the content of the message being signed [MOV96]. Digital signatures provide authentication, integrity and non-repudiation services. The process of digital signature generation and verification is depicted in the figures below.

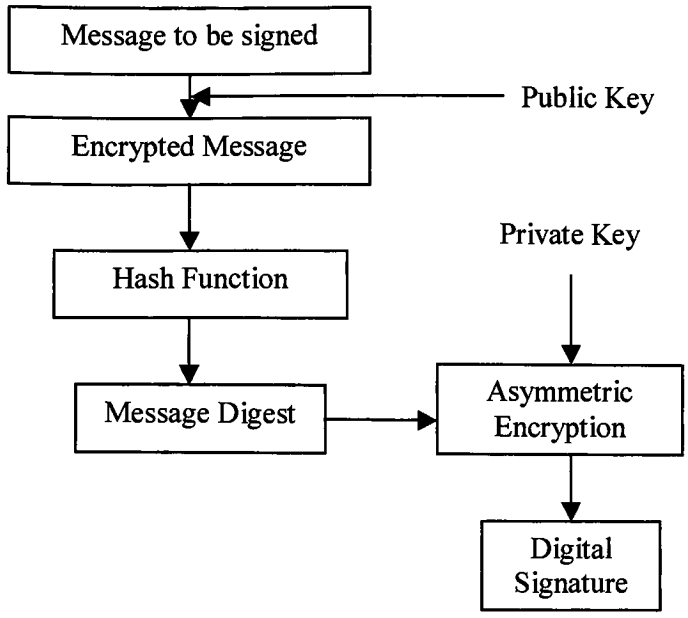


Fig 3.4: Generation of a Digital Signature

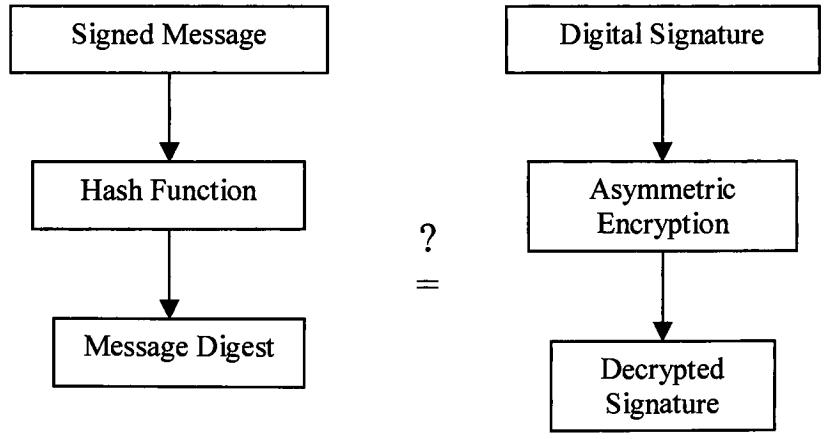


Fig 3.5: Verification of a Digital Signature by the receiver, by comparing the message digest with the decrypted signature

The message, which is to be signed, is encrypted using the recipient's public key. A hash function is applied to this encrypted message to generate a message digest. The hash function has two properties – one that it is a one-way function and secondly it produces a unique message digest for every message it is applied to. Message digest is typically smaller in length than the original message, because the hash function compresses the message from an arbitrary length to a smaller fixed length [RSA00]. A transformation (encryption algorithm) is applied to the message digest using the senders private key (meaning that the sender has now signed the message, since his private key is unique) and a digital signature is generated. The encrypted message and the digital signature are both transmitted to the receiver.

The receiver first decrypts the message using his private key. He then applies the same one-way hashing function to this message to generate the message digest. (Information about which hashing function used is also transmitted by the sender.) The recipient then decrypts the digital signature to get the message digest. If the two message digests generated match, it means that the message has not been modified (integrity), the sender is identified (authentication) and it cannot to be repudiated in the future since the message digest was encrypted using the sender's private key.

3.5 Discrete Logarithm Problem (DLP)

The discrete logarithm problem applies on certain algebraic structures called groups. A group is a finite or infinite set of elements (finite in cryptographic applications), along with a binary or group operation, which satisfies certain mathematical properties.

DLP forms the basis of most public key cryptosystems because it is a hard mathematical problem to compute. The running time for the algorithms used to compute the discrete logarithm is approximately $O(\sqrt{g})$ (square root of the order of group g) [ODL99]. The order of a group is the number of elements in the group, which is about 2^m (m is in the range of 160 -560) for cryptographic applications. If m = 160, the assumption is that the DLP cannot be solved any faster than $\sqrt{2\text{^}160}$, which is about 10^{24} group operations. This would require about 1026 computer instructions. A MIPS (Millions of Instructions

per Second)-year (MY) is about 3 x 10^{13} instructions, so to solve the discrete logarithm it would require 10^{12} MY [ODL99].

Mathematically, DLP is represented as follows

Consider an element a in group G. Now if a is multiplied n times to itself it gives another element b in G.

$a \times a \times a \ldots\ldots \times a$ (n times) $\bmod p = b = a^n \bmod p$.

The discrete logarithm problem is to find the inverse n given a, b and p.

Example

For a = 6, n = 9, p = 11,

$b = a^n \bmod p = a\,(a^2)^2)^2 \bmod p$

with four multiplications:

$b = 6\,(6^2)^2)^2 \bmod 11 = 6\,(36)^2)^2 \bmod 11$

$= 6\,(3)^2)^2 \bmod 11 = 6(9)^2 \bmod 11$

$= 6\,(81) \bmod 11 = 6(4) \bmod 11$

$= 24 \bmod 11 = 2$

However finding an n such that $2 = 6^n \bmod 11$ is difficult.

Various researchers have investigated the discrete logarithm problem [COP84, ODL84] and it has been reported to have expected running times similar to those of the best factoring algorithms. Rivest [RIV92a] has analyzed the expected time to solve the discrete logarithm problem both in terms of computing power and cost. The best known schemes for solving the discrete logarithm problem are the index-calculus methods and collision search methods [RSA00]. Index calculus is a sub-exponential time algorithm whereas collision search takes fully exponential time to solve the DLP.

3.6 Key Management

Key management involves secure generation, distribution and storage of keys. Exchanging secret key securely is the main problem of secret key cryptosystems [RSA00]. Various methods have been devised for the distribution of secret keys. There

are 2 automated methods for key distribution, which are discussed, in the American National Standards Institute (ANSI) standard X9.17, namely the "Financial Institution Key Management" and the "Diffie/Hellman key exchange".

3.6.1 Diffie Hellman (DH) Key Exchange

The steps involved in Diffie Hellman key exchange protocol are as follows.

Assume Alice and Bob are the two people in the communication and they want to generate a secret key.

Both select a prime number p and a primitive root $g \bmod p$. Both these parameters are made public. The root g is such that for any number n from 1 and p-1, there exists an integer k such that $n = g^k \bmod p$

1. Alice selects a random value a from between 1 and $p-1$
2. Alice computes $g^a \bmod p$ and sends it to Bob
3. Bob selects a random value b from between 1 and $p-1$
4. Bob computes and sends $g^b \bmod p$ to Alice
5. Alice then computes $(g^b \bmod p)^a = g^{ab} \bmod p$
6. Bob computes $(g^a \bmod p)^b = g^{ab} \bmod p$, which is the shared secret key.

The strength of the Diffie-Hellman key exchange protocol lies in the fact that given g, p, ($g^a \bmod p$) and ($g^b \bmod p$) it is computationally infeasible to find $g^{ab} \bmod p$ [RSA00]. The DH key exchange is based on the DLP and has similar running times as the DLP [MAU94].

3.7 Certificates

In public key cryptography, users use certificates in order to obtain each other's public key securely. A certificate is document, which cannot be forged, containing information about the user's public key. The certificate can be obtained from a central issuance agency called Certifying Authority (CA) [RSA00]. CA is a centralized administration

such as a company which issues certificates to its employees, or a university which issues certificates to its faculty, students and staff. It maintains a public key ring server which stores the public keys securely. The CA either gets its certificate from another higher authority or may make its public key, public.
A certificate contains the digital signature of the CA, the CA's public key and the user's public key. Certificates also contain other information such as the time for which it is valid. Certificate formats are discussed in [KEN93].

Certificates prevent impersonation. A sender encrypts the message using the recipient's public key and sends it to him/her. The recipient uses the CA's public key to decode the digital certificate attached to the message, verifies it as issued by the CA and then obtains the sender's public key and identification information held within the certificate. With this information, the recipient can now send an encrypted reply. If messages are exchanged between a pair of users quite often, the more they trust each other, and certificates need not be enclosed and verified every time [RSA00]. If two people send messages to each other every day, they only need to enclose certificates the first time. After some exchanges the users store each other's public key and certificates are no longer required.

Chapter 4

Mathematical Background

Introduction

Koblitz [KOB87] and Miller [MIL86] independently proposed elliptic curves defined over a finite group of elements known as a finite field, for cryptographic applications. The security of the ECC is based on the intractability of the DLP over finite groups of numbers or finite fields. The DLP over group of points of an elliptic curve defined over a finite field (Elliptic Curve DLP) is significantly more difficult than the DLP over ordinary finite groups of numbers [CLH00]. This is because in ordinary groups of numbers solving the DLP is simply computing the inverse of modular exponentiation [Sec. 3.5]. For additional security in these groups, large exponentiations have to be computed, which is a very expensive operation in terms of resources required. The structure of elliptic curve groups provides more flexibility in selecting the parameters of the cryptosystem which simplifies computations and can lead to efficient implementations [LM95].

There are finite number of points on an elliptic curve defined in the X-Y plane, along with a special point at infinity O, which is also called as the origin point or the additive identity. The elliptic curve points can be counted and an addition operation can be defined on them, such that adding any two points results in a third point on the curve. These points, along with the addition operation form what is known as the elliptic curve group. The selection of points, for elliptic curve cryptographic implementations is done from a group of numbers, which form a finite field, and so elliptic curve groups are said to be defined over finite fields. Finite fields and groups obey certain mathematical properties which are discussed later on in this chapter.

This chapter begins with an introduction to some mathematical structures, which are necessary in order to understand well groups and fields, the two most important concepts required in order to implement an ECC. More information on finite fields and groups is

found in [LHC01, MH93]. The points on the elliptic curve can be represented in certain formats such as a polynomial basis or optimal normal basis representations. These representations are discussed in this chapter. Also included are the finite field arithmetic operations required in the elliptic curve scalar multiplication process, which forms the heart of the ECC.

4.1 Some basic structures of Algebra

Algebra is the branch of mathematics that defines the basic arithmetic relationships using variables. It studies the properties of operations on algebraic structures. Structures of algebra form a collection of objects and operations which can be used to calculate and solve equations. These objects can be numbers, polynomials or geometrical figures.

4.1.1 Identity Element

Every elliptic curve group contains an identity element with respect to the operation defined on that group.

An element e of a set X is called an identity element of X relative to any operation, say *, (notation for the operation defined on set, such as addition or multiplication) on X if

$$e * x = x * e = x$$

for all $x \in X$

Example: The number 1 is the identity element for the set of integers, under the multiplication operation as any number multiplied by 1 gives the same number.

4.1.2 Groups

The points on an elliptic curve form an elliptic curve group with the addition operation.

If * is an associative operation on set X, then {X ;*} (set X with the operation *) is known as a **semigroup**.

Example

Let a, **b**, c $\in$ X and let '*' be an operation on X.

If a * (b * c) = (a * b) * c; then {X ;*} is a semigroup.

If X contains an identity element relative to * then {X ;*} is called a **monoid.**

Example

Let X be a semigroup and let a be an element in X. If $a*e=e*a=a$; then e is an identity element and X is a monoid with operation *.

A **group** is a monoid in which every element is invertible; i.e., a group is a semigroup {X ;*}, with identity in which every element is invertible or has an inverse in the same group.

Properties of a group

Let {X ;*} be a group and let e be the identity element of X relative to *.

Let a and b be elements of X. Then the group X is said to have the following properties.

1. **Closure** – a * b = c $\in$ X. For every operation on elements in X, the resulting element in also in X.
2. **Associative** – The group operation * is associative; i.e., a * (b * c) = (a * b) * c for c in X.
3. **Identity** - e * b = b * e = b. (e is the identity element)
4. **Inverse** – For every element in X, there exists an inverse in X. a, $a^{-1} \in X$.
   Also a * a^{-1} = a^{-1} * a = e.

Example: The set of real numbers with the addition operation form a group, since it satisfies all the properties of a group.

Closure - Addition of any two real numbers, results in a real number

Associative – Addition is an associative operation;

(2.3 + 4.0) + 6.57 = 2.3 + (4.0 + 6.57)

Identity – Zero is the identity elements in the set of real numbers. Adding any number with 0 is the same number.

Inverse – The inverse of any number in the set of real numbers is the negative of that number. 4.67 + (-4.67) = 0 (identity element).

If X has a finite number of elements then X is known as a **finite group.**

The following two definitions (abelian group and rings) lead into the definition of a field, over which the elliptic curve cryptosystem is defined.

Abelian group

A group {X ;*} is said to be an abelian group if the operation * is commutative; i.e.,

$a * b = b * a$

The points on an elliptic curve form an abelian group with the addition operation [SAE96], as addition is a commutative operation.

4.1.3 Rings

Let X be a non-empty set, and let + and * be two operations on X. Then {X; +, *} is said to be ring if

1. {X ;+} is an abelian group.
2. {X;*} is a semigroup; and
3. The operation * is distributive over +. For example - (a + b) * c = (a * c) + (b * c) and c * (a + b) = (c * a) + (c * b).

If in the ring {X; +, *}, the operation * is commutative then it is known as a **commutative ring.**

4.1.4 Finite Field

A **field** {X; +, *} is a ring in which the * operation is commutative and all non-zero elements have multiplicative inverses, i.e., for every $a \neq 0$, $a \in X$ there exists an element $a^{-1} \in X$ such that

$$a * a^{-1} = a^{-1} * a = 1.$$

If the number of elements in the field is finite, it is called finite field. Elliptic curve groups are defined over finite fields. ECC rely on the difficulty of computing the discrete logarithm over finite fields.

4.1.5 Some other definitions

Order of a Point

The order of an element $a \in X$, is the smallest positive integer t such that $a^t = e$ (identity element). The elliptic curve scalar multiplication involves computing kP, where P is a point on the curve and k is a random integer, resulting in another point on the curve ($Q = kP$). If the order of the point P is a prime number of n bits, then computing k, knowing kP and P takes roughly $2^{n/2}$ operations. If point P is 160 bits long, then to find k, the number of operations needed is about 2^{80}. If someone does a billion operations per second, this takes about 38 million years [ENG04].

Order of the Curve

The order of the curve is the total number of points on the curve. Knowledge of the order of the curve is helpful in determining the security level of the cryptosystem [ROS99]. If the order contains a large prime factor, then it is considered to be more secure against certain attacks like the Pohlig-Hellman algorithm [ENG99]. However, finding this number for random curves is difficult and for most applications it is acceptable to not know the order of the curve [ROS99]. If the group order is large enough then most attacks do not work [ENG99]. However, for advanced nuclear-grade crypto it is essential to know the number of points on the curve.

It is suffice for most applications, that the number of points satisfy ***Hasse's theorem*** which states that -

Given a field Fp, the order of the curve N will satisfy the equation

$$|N - (p+1)| \leq 2\sqrt{p} .$$

If n is 2^{140}, there is a reasonable probability that the order of the curve will contain a prime factor of the order of 2^{112} or greater [ROS99]. For maximum security, curves having order with prime factors close to the size of the field i.e., 2^{n-2} are chosen, where the size of field GF(2^n) is 2^n.

Characteristic of a field

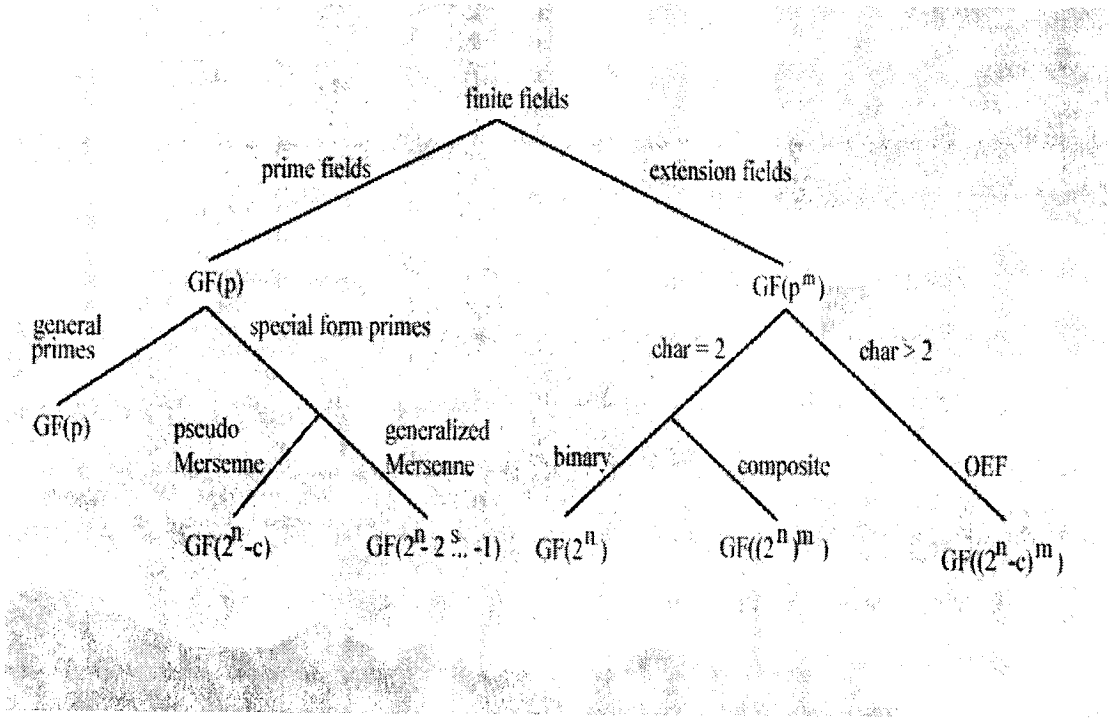
The characteristic of a field is the smallest possible integer p such that adding 1 to itself p times yields 0. For a field represented as GF (p^n), p is the characteristic of the field. When n = 1, the resulting field is known as prime field and the points on the elliptic curve are selected from between 0 and p-1. Fields with characteristic 2, GF (2^n) are also standardized for implementing ECC as an alternative to prime fields.

Generator of a field

Finite fields over which elliptic curves are defined are a cyclic group of numbers. All the elements in the group can be generated from a single element called generator of the group or the primitive element. An element g is said to be a generator in X if every element in X can be expressed as g^t for some integer t.

4.2 Finite Field Options

Finite fields are also known as **Galois field,** represented as GF (q), where q is the number of elements in the finite field. Finite fields exist only for $q = p^m$, where p is a prime number and m is a positive integer. The figure below shows the different types of finite fields over which ECC can be defined.



OEF – Optimal Extension Fields

Fig 4.1: Finite Field Options [PAR99]

In cryptographic applications, two kinds of fields are commonly used [IEEE99], because these fields are extremely well studied and a number of algorithms [IEEE99] have been proposed for efficient arithmetic in these fields. Also both these fields are included in the standard draft for public key cryptography (IEEE P1363) for implementing an ECC.

1. Prime fields GF (p) where p is large prime number.
2. Binary fields GF (2^m) where m is large integer and is mostly a prime number.

The following provides an explanation of prime fields and binary fields. Section 4.5 discusses the factors, which influenced the selection of binary field over prime field for this particular implementation.

4.2.1 Prime Field

A prime field GF (p) consists of a finite group of numbers between 0 and $p-1$ with the addition and multiplication operation. All operations in GF (p) are performed modulo prime p, resulting in an element in the same field. All the elements of the prime field can be generated from a single element g in GF (p) such that $g^i \bmod p \in$ GF (p). (g is known as the generator of the field GF (p)), where $0 \leq i \leq p-1$.

4.2.2 Binary Field

A binary field is represented as GF (2^m) and consists of 2^m elements. For any prime power there exists only one finite field, so for a particular m two representations of GF (2^m) are isomorphic [WM03]. The most common forms of representing elements of the binary field are polynomial basis and optimal normal basis.

4.3 Polynomial Basis

A polynomial f (x) over a field GF (p) is represented as follows:

$$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots\ldots\ldots\ldots + a_{n-1} x^{n-1} + a_n x^n$$

where $a_0, a_1, a_2 \ldots\ldots a_n$ are known as coefficients of the polynomial and x is the variable which is a complex number.

The polynomial f (x) can be represented in vector form as $\{a_0, a_1, a_2 \ldots\ldots a_n\}$.

Example – $4x^4 + 5x^3 + x + 1$ is a polynomial, represented in vector form as {4, 5, 0, 1, 1} All operations are performed on the coefficients of the polynomials.

A field element in polynomial basis can be generated in GF (2^m) with coefficients in GF (2) (i.e., 0 and 1) with an irreducible polynomial (A polynomial is said to be irreducible in GF (q) if it cannot be factored into a product of lower degree polynomials in GF (q)) of degree m.

Operations in Polynomial Basis

In the elliptic curve scalar multiplication process, four arithmetic operations in any field are required. These include field addition, field multiplication, field squaring and field inversion.

Addition and subtraction of two polynomials in GF (2^m) is the exclusive-OR (XOR) operation; because addition and subtraction modulo 2 is XOR.

$1 + 1 \bmod 2 = 0$

$0 + 1 \bmod 2 = 1$

$0 + 0 \bmod 2 = 0$

$\{a_0, a_1, a_2 \ldots a_{n-1}, a_n\} + \{b_0, b_1, b_2 \ldots b_{n-1}, b_n\} = \{c_0, c_1, c_2 \ldots c_{n-1}, c_n\}$.

where $c_i = a_i$ xor b_i

Example

$(x^4 + x^3 + x + 1) + (x^4 + x^2 + x + 1)$

Coefficients of equal powers of x in the two polynomials are XOR ed with each other.

Therefore,

$x^4 \quad 1 + 1 = 0$

$x^3 \quad 1 + 0 = 1$

$x^2 \quad 0 + 1 = 1$

$x \quad 1 + 1 = 0$

$x^0 \quad 1 + 1 = 0$

which is $x^3 + x^2$

Multiplication in polynomial basis is simply shift left and XOR (add).

For example

$(x^4 + x) * (x^2 + x + 1)$

$= x4 * (x^2 + x + 1) + x * (x^2 + x + 1)$

$= x^6 + x^5 + x^4 + x^3 + x^2 + x$

The exponents get added in the multiplication process. The highest exponent keeps increasing, so the multiplication has to be done modulo an irreducible polynomial.

Similarly inversion can be computed in polynomial basis using various known algorithms such as Fermat's Little Theorem (6.4.1) and the Extended Euclidean Algorithm [ROS99].

4.4 Optimal Normal Basis (ONB)

Unlike polynomial basis, optimal normal basis representation is not very common in ECC implementations, because the representation is somewhat complex because it involves double exponentiation. However it has been used for efficient implementations of cryptosystems, especially in hardware [LMW00], since exponentiation in ONB is simply the left rotation of the coefficients. This considerably speeds up the inversion operation, which involves exponentiation, and leads to efficient hardware implementations.

As seen in 4.3 an element in polynomial basis is represented as

$$a = a_0 + a_1 x + a_2 x^2 + \ldots\ldots\ldots\ldots + a_{n-1} x^{n-1}$$

A normal basis can be formed by using the set

$$\{ a^{p^{n-1}} \ldots\ldots a^{p^2}, a^p, a \}$$

Any element in field GF (2^m) can be represented in normal basis as

$$b = b_{m-1} a^{2^{m-1}} + \ldots\ldots + b_2 a^{2^2} + b_1 a^2 + b_0 a$$

or $b = \sum_{i=0}^{m-1} b_i a^{2^i}$

where b_i in GF (2).

When two elements in normal basis are multiplied the result obtained is known as the lambda matrix [MOV89]. An "optimal" normal basis (ONB) has the minimum number of non-zero terms in the lambda matrix. This value was found out to be $2^m - 1$ [ABV89]. There are two types of optimal normal basis [MOV89, ROS99]. The value of m in the field GF (2^m) determines if an ONB exists for that field and the type of ONB.

1. **Type I ONB**

   For a Type I ONB m + 1 must be prime and 2 must be primitive in the set of integers 0 to m + 1, which means that 2 raised to any power in the range 0 ... m-1 modulo m + 1 must result in a unique integer in the range 1...m.

2. **Type Π ONB**

   To form a Type Π ONB, (2m + 1) should be prime and either 2 is primitive in the set of integers 2m + 1 or 2m + 1 ≡ 3 mod 4. (≡ means congruent).

Addition and subtraction in ONB is similar to that defined in polynomial basis. Multiplication is complex and involves a series of steps [ROS99].
Exponentiation in ONB is very efficient since it can be performed as a simple rotate; so squaring and inversion (most inversion algorithms involve exponentiation operations) are easier using the ONB representation.

This thesis focuses on polynomial basis, so the details of ONB arithmetic are not discussed in detail. The reasons for the selection of polynomial basis over ONB for this particular implementation are explained in the following section.

4.5 Implementation Options

The selection of the following greatly influences the implementation and the strength of the resulting cryptosystem [CERT98].

1. Selection of the underlying finite field.
2. Selection of the representation of the elements in the finite field
3. Selection of the elliptic curve.

Selection of finite field

Prime fields are very popular in cryptographic systems. Arithmetic operations such as addition and subtraction are very easy and can be performed with limited resources. However, inversion in prime fields is slow, and so is multiplication. There are not many efficient multiplication algorithms proposed for implementation in prime fields as there are for binary fields. Using GF (p) generally requires a coprocessor for performing the modular operation, which increases performance but the addition of a coprocessor increases costs by 20 – 30 % [CERT98]. Another major disadvantage of using prime fields for cryptosystems is that multiplication needs to be performed with long numbers (160-2048 bits) on processors (8 – 64 bits) with short word length [PAR99]. However, as far as the strength of the cryptosystem is concerned there have been no discoveries to date, which suggest that using prime fields over binary fields for elliptic curve cryptosystems makes it any easier or harder for attack [CERT00]. The selection of a particular finite field depends solely on the implementation platform and choice.

Binary fields are well researched and a number of algorithms [DLA00, GP97] have been proposed for arithmetic in binary fields. The major factor for the choice of binary fields is that the coefficients are 0 and 1, which makes it relatively easy for coding. Also it is very well suited for hardware implementations. [CERT00] also mentions that a hardware implementation using binary fields offer significant performance and die size advantages over prime field implementations, since a coprocessor optimized for GF (2^m) arithmetic would take up less space and offer increased performance levels than the available

crypto-coprocessors for prime field arithmetic. This is because arithmetic in prime fields involves computation with very large numbers which requires more hardware resources.

Selection of representation of field elements

When using bimary fields, the field elements are commonly represented in polynomial basis and optimal normal basis as discussed previously. Similar to finite field selection, the representation of elements does not make the cryptosystem any more susceptible to attack. Also one type of representation can be easily converted into the other type of representation. Polynomial basis representation is mathematically less complex than ONB as seen from section 4.3 and 4.4. Representation in ONB requires two exponentiations. However some hardware implementation have shown that using ONB is more efficient since operations are computed by exclusive-OR, shift and rotate which are fast and less resource consuming [LL02]. Polynomial basis have been proven to be faster in some software implementations [Sec. 2.1].

[DAH97] showed that a combination of both polynomial basis and optimal normal basis can be used to achieve maximum efficiency by exploiting advantages of both.

Selection of Elliptic Curves

The National Institute of Standards and Technology (NIST) recommend a set of elliptic curves for implementation. Non-super-singular curves of the form $y^2 + xy = x^3 + ax + b$ are recommended because a recent attack on super-singular curve ($b = 0$) has made them less useful in practice [LM95]. Special kinds of curves called Koblitz curves [KOB92] have been proposed which have shown to result in faster elliptic curve arithmetic implementations.

There are advantages in using standard elliptic curves because they provide efficient implementation, and offer less bandwidth and storage requirements. However standard elliptic curves are being well studied by cryptanalyst and are becoming less secure [HIT02]. Generating a random elliptic curve is time consuming [ROS99], but leads to more secure implementations. In order to increase the immunity of a fixed curve, i.e., in order to make in more secure, the order of the curve needs to be increased (i.e., the

number of points on the curve should be increased). So as there are more points this which means more resources are needed to implement it in hardware.

The implementation reported in this thesis uses binary field with polynomial basis representation and random elliptic curves. Arithmetic in binary fields is less complex and lends itself easily to coding because the coefficients are 0 and 1. Polynomial basis representation is common and is easier to implement than ONB. Also polynomial basis exists for every prime power whereas ONB exists only for certain primes. Random elliptic curves offer more security if they are properly chosen, as these are not documented and well-studied by cryptanalysts. Also most attacks are not possible on random curves [HIT02]. Some researchers suggest that for extremely critical operations standard elliptic curves should be used as they are proven to offer good security levels and generating random curves may compromise the security of the system [ROS99]. It is very important to ensure that the order of the curve be large and divisible by a large prime when selecting an elliptic curve, so that the ECC becomes immune to most known attacks.

Chapter 5

Introduction to Elliptic Curves

An elliptic curve over a group of numbers is a set of solutions of the equation

$$y^2 + sxy + ty = x^3 + ax + b \quad \text{---------- (5.1)}$$

together with a point at infinity O. This equation is known as the "**Weierstrass**" equation [ROS99].

s, t, a, b, x, y belong to the group over which the elliptic curve is defined. This maybe the set of real numbers or a finite field for cryptographic applications. The point at infinity O is also the identity point of the elliptic curve. The points on an elliptic curve form an abelian group with the point addition operation [Sec. 4.1.2].

Elliptic curves for cryptographic applications are defined over finite fields, meaning the points on the elliptic curve belong to a finite field. Elliptic curves can be geometrically represented over a group of real numbers and so they are discussed first to provide a good understanding followed by a discussion on elliptic curve groups over finite fields. Point addition and point doubling are discussed in each case because these are the two computations required in the scalar multiplication process.

5.1 Elliptic Curves over Real Nnmbers

A simpler form of the elliptic curve equation is $y^2 = x^3 + ax + b$, by taking s and t = 0.

By choosing different values of a and b from the group of real numbers, the elliptic curve can be plotted in the x-y plane. An important criterion for elliptic curves over real number is that $x^3 + ax + b$ should not have any repeated factors or equivalently $4a^3 + 27b^2 \neq 0$ [CERT00].

Example

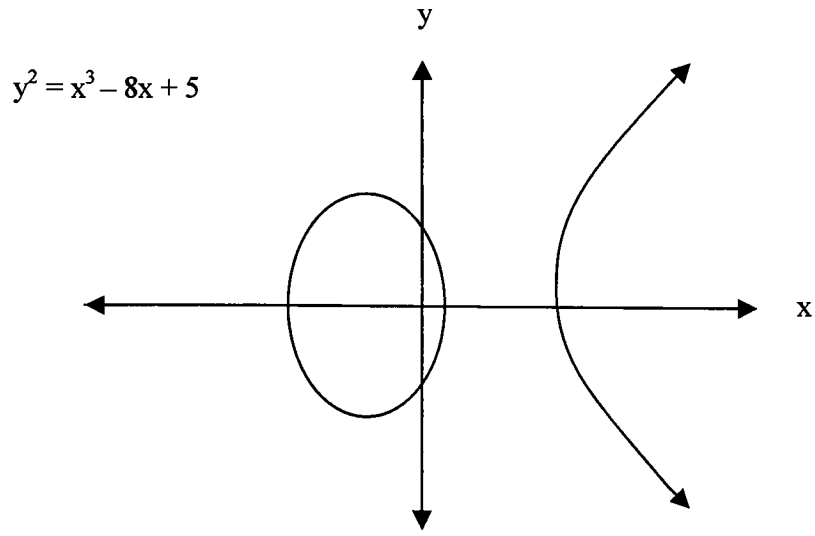


Fig 5.1: Elliptic Curve over real numbers

Addition of 2 points on the elliptic curve over real numbers is represented graphically as follows (Fig 5.2). P and Q are the two points to be added. To do this geometrically, a line is drawn joining points P and Q, intersecting the curve at exactly one more point -R. A perpendicular to the x-axis is drawn from this point to intersect the curve at another point R. The point R is the result of adding points P and Q.

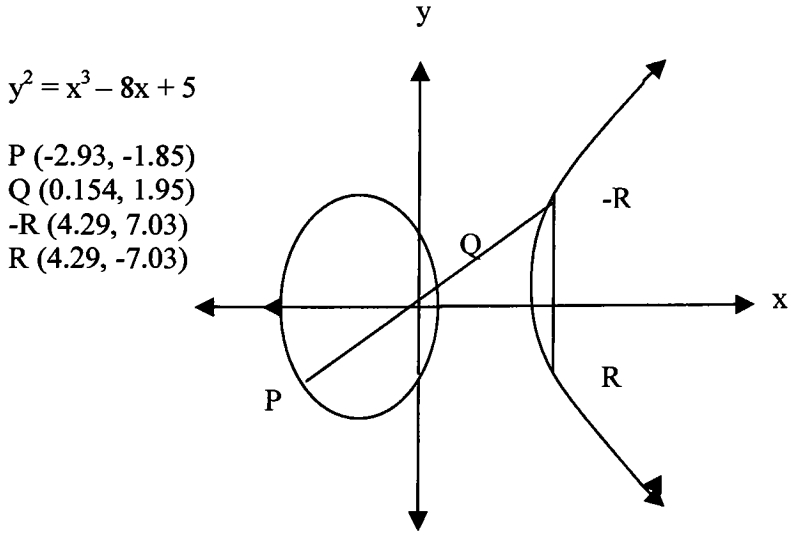


Fig 5.2: Point Addition

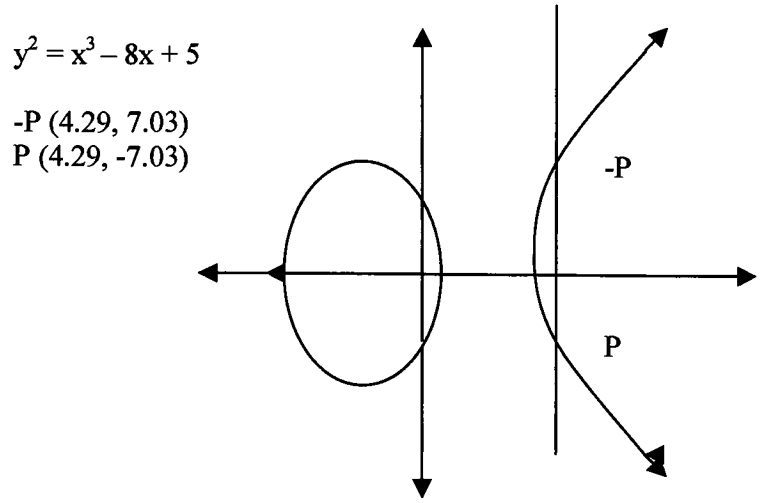


Fig 5.3: Adding points P and –P is the point at infinity O.

If the two points to be added are negative of each other, then the line passing between the two points intersects the curve at O, which is the point at infinity on the curve. It is the identity element (Sec. 4.1.1) of curve for the addition operation since the points on the elliptic curve form an abelian group (Sec. 4.1.2) with the addition operation. This also proves that point O lies on the curve.

Point Doubling

The geometrical method for doubling a point on the curve consists of drawing a tangent at the point (P), which intersects the curve at another point (-R). A perpendicular drawn at this point intersects the curve at another point, which is 2P. This is represented as shown in the figure below.

$y^2 = x^3 - 8x + 5$

P (0.308, 1.61)
-R (5.19, -10.2)
R (5.19, 10.2)

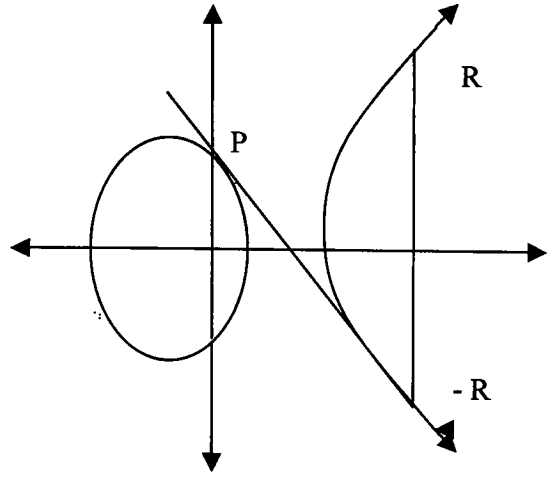


Fig 5.4: Point Doubling

$y^2 = x^3 - 8x + 5$

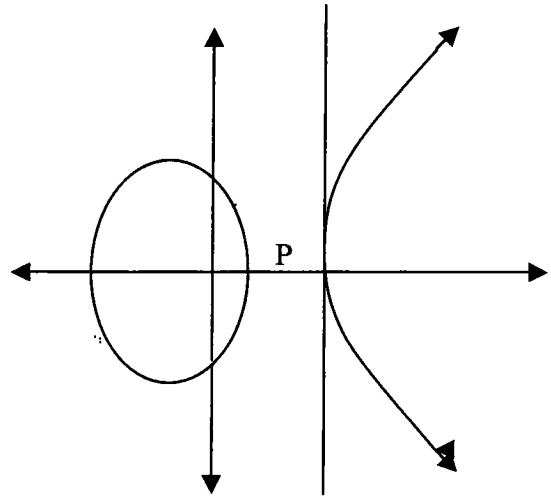


Fig 5.5: Doubling a point on the X-axis gives the point at infinity O.

Now, if the point to be doubled lies on the X-axis then a tangent drawn at that point intersects the curve at O, the point at infinity.

Mathematical Formulae for Point Addition

Mathematically, addition can be represented as follows:

Let there be two points on the curve $P(x_p, y_p)$ and $Q(x_q, y_q)$. Points P and Q should not be negative of each other.

If s is the slope of the line between P and Q, then $s = (y_q - y_p) / (x_q - x_p)$

Let $R (x_r, y_r) = P + Q; P \neq -Q$

Then

$$x_r = s^2 - x_p - x_q$$

$$y_r = - y_p + s (x_p - x_r)$$

Doubling the point P on the curve is mathematically expressed as follows –

When y_p is not 0,

Let $R (x_r, y_r) = 2 P$.

$$s = (3 x_p^2 + a) / (2 y_p)$$

$$x_r = s^2 - 2x_p$$

$$y_r = - y_p + s (x_p - x_r).$$

5.2 Elliptic Curves over Finite Fields

Arithmetic over real numbers can lead to rounding errors. Elliptic curves defined over finite field are considered a good approach for implementing cryptosystems [SIM92], as arithmetic over finite fields is efficient and well suited for cryptographic applications [YOR92]. Most commonly used fields for cryptographic applications are prime fields and binary fields [IEEE99].

5.2.1 Elliptic Curves over Prime Fields

Elliptic curve groups are defined over prime fields GF (p) where p is a large prime. The elliptic curve equation remains the same as in real numbers i.e.,

$$y^2 \bmod p = x^3 + ax + b \bmod p$$

where a, b are chosen from the prime field GF (p). The selection should be such that the right hand side of the equation should have no repeated roots ($4a^3 + 27b^2 \neq 0$). Then the

set of solutions to the above equation form an elliptic curve over the prime field GF (p). There are finitely many points on such an elliptic curve.

The number of points on an elliptic curve must satisfy **Hasse's theorem** which states that – Given a field, Fp, the order of the curve N will satisfy the equation

$| N - (p + 1) | \leq 2\sqrt{p}$

Example

Consider the prime field with GF (11); p = 11 and let the elliptic curve equation be $y^2 = x^3 + x + 6$; where a = 1 and b = 6.

There are 12 points lying on this elliptic curve which satisfy the elliptic curve equation. (2, 4) , (2, 7) , (3, 5) , (3, 6) , (5,2) , (5, 9) , (7, 2) , (7, 9) , (8, 3), (8,8), (10,2) , (10,9) which means there are 13 points altogether on the elliptic curve along with the point at infinity O.

Since there are a finite number of points, this does not represent a curve which can be plotted in any definite shape; but an important point to note is that there are two points for every value of x. So even if the plot is random there is symmetry about the X-axis.

Mathematically the equation for point addition and point doubling remain the same; the only difference being that now the arithmetic is computed by taking modulo p.

Adding 2 points on the elliptic curve over GF (p)

Let $R (x_r, y_r) = P + Q, P \neq -Q$.

Then

$$s = (y_q - y_p) / (x_q - x_p) \bmod p$$

$$x_r = s^2 - x_p - x_q \bmod p$$

$$y_r = - y_p + s (x_p - x_r) \bmod p$$

Doubling the point P on the curve is mathematically expressed as follows

When y_p is not 0,

Let R $(x_r, y_r) = 2$ P.

Then

$$s = (3 x_p^2 + a) / (2 y_p) \bmod p$$

$$x_r = s^2 - 2x_p \bmod p$$

$$y_r = - y_p + s (x_p - x_r) \bmod p.$$

5.2.2 Elliptic Curve Groups over binary extension fields

The non-super singular elliptic curve equation over binary fields is given as

$y^2 + xy = x^3 + ax + b$, where a, b are selected from the field GF (2^m) and $b \neq 0$.

Elliptic curve groups over binary fields GF (2^m) can be represented in either polynomial representation or normal basis representation as discussed in 4.4 and 4.5.

The points on the elliptic curve are such that x and y lie in the binary field.

Example

Consider the field GF (2^4) with irreducible polynomial $f(x) = x^4 + x + 1$.

a = (0010) is the generator of the field and the powers of a are

$a_0 = (0001)$ $a_1 = (0010)$ $a_2 = (0100)$ $a_3 = (1000)$

$a_4 = (0011)$ $a_5 = (0110)$ $a_6 = (1100)$ $a_7 = (1011)$

$a_8 = (0101)$ $a_9 = (1010)$ $a_{10} = (0111)$ $a_{11} = (1110)$

$a_{12} = (1111)$ $a_{13} = (1101)$ $a_{14} = (1001)$ $a_{15} = (0001)$

Let the elliptic curve equation be $y^2 + xy = x^3 + g^2 x^2 + 1$, where $a = g^2$ and $b = 1$

There are 15 points that satisfy the above equation which are

$(0, 1)$ $(1, a^{12})$ (a^5, a^{14}) (a^9, a^{10}) (a^6, a^3) $(a^{12}, 0)$ (a^5, a^{11}) (a^1, a^{12})

$(1, a^7)$ (a^5, a^9) (a^9, a^1) (a^6, a^7) (a^{12}, a^{12}) (a^5, a^{13}) (a^1, a^6)

Mathematical representation of addition and doubling on elliptic curve over GF (2^m)

The formulae for point addition and point doubling are slightly different for binary fields as compared to real numbers and prime fields.

Adding two points P and Q

Let R $(x_r, y_r) = P + Q$, $P \neq -Q$.

Then

$$s = (y_q - y_p) / (x_q - x_p)$$

$$x_r = s^2 + s + x_p + x_q$$

$$y_r = y_p + x_r + s(x_p + x_r)$$

Doubling the point P on the curve is mathematically expressed as follows –

When x_p is 0, then $2P = 0$

When x_p is not 0, then

Let R $(x_r, y_r) = 2$ P.

Then

$$s = x_p + y_p / x_p$$

$$x_r = s^2 + s + a$$

$$y_r = x_p + (s + 1) * x_{r\backslash}$$

5.3 Elliptic Curve (EC) Scalar Multiplication

The elliptic curve scalar multiplication is the heart of the elliptic curve cryptosystem. Scalar multiplication involves computing kP; where k is an integer and P is a point on the elliptic curve. This process is a one-way function and computing its inverse takes fully exponential time.

The following figure depicts the hierarchy in computation of elliptic curve scalar multiplication.

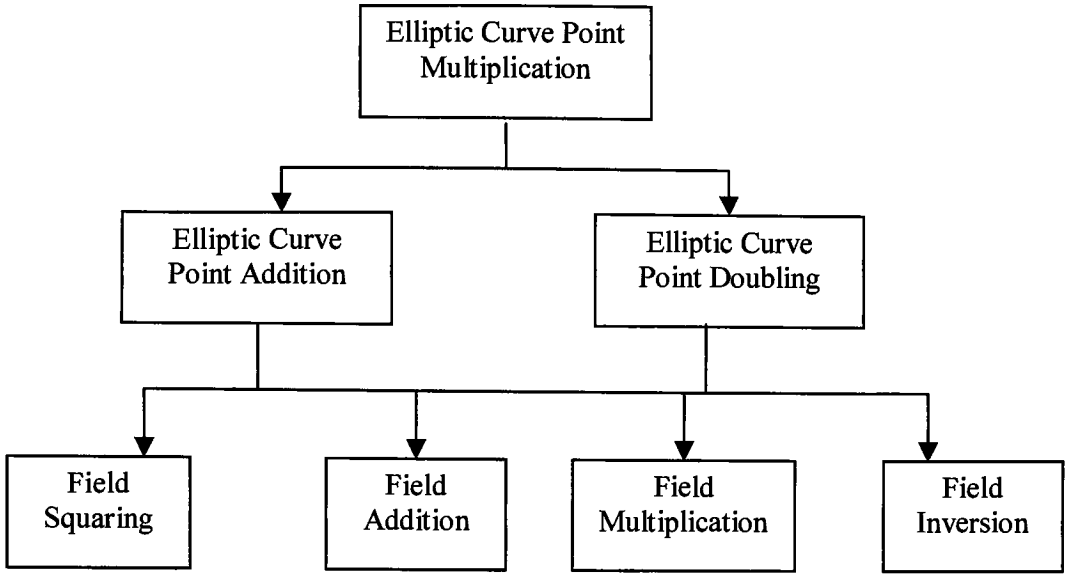


Fig 5.6: Hierarchy of arithmetic in EC scalar multiplication

The elliptic curve scalar or point multiplication is computed by repeated point additions and point doublings. Adding and doubling a point over a field requires the following field operations – field addition, multiplication, inversion and squaring.

To calculate kP, the point P is added to itself k times

$kP = P + P + P \ldots + P$ (k times).

This can also be represented a series of point additions and doublings.

$kP = P + \ldots(2(2(\ldots P + 2(P + 2(P + 2P)))$

Example: If k = 15 then

$15P = P + 2(P + 2(P + 2P))$.

Various algorithms have been proposed for the elliptic curve scalar multiplication operation; which are discussed in Chapter 6.

5.4 Elliptic Curve Discrete Logarithm Problem (ECDLP)

The strength of the elliptic curve cryptosystems lies in the intractability of the elliptic curve discrete logarithm problem [MEN95]. Like the discrete logarithm problem discussed in Sec. 3.5, the one-way function for elliptic curve cryptosystems is computed by the elliptic curve multiplication operation, which is finding kP where k is a random integer and P is a point on the curve. This results in another point Q on the curve. The ECDLP is based on the fact that knowing P and Q, there exists no sub exponential algorithm to find k [ROS99]. k is the discrete logarithm of Q to the base P.

There has been considerable research into solving the elliptic curve discrete logarithm problem. Except for some cases like the super singular curves or when the order of the point P is not prime, there has been little success. The fastest known technique is known as the Pollard Rho method [HM04]. It has a running time of $\sqrt{\Pi n/2}$ elliptic curve operations, which is still exponential n. The table below [ECC9] shows the time required to solve the ECDLP in MIPS years using the Pollard Rho method.

Key Sizes	MIPS Years
150	$3.8 * 10^{10}$
205	$7.1 * 10^{18}$
234	$1.6 * 10^{28}$

Table 5.1: Computational Effort for Cryptanalysis of DLP using Pollard Rho.

There are also no index-calculus methods for solving the ECDLP as there are for DLP [Sec 3.5, HM04], because index calculus requires certain mathematical properties to be satisfied which are absent in elliptic curve groups, making them resistant to such attack. The algorithms for solving ECDLP are much more difficult and time consuming [HM04], which is why the elliptic curve cryptosystem can achieve similar levels of security with smaller key lengths than public key schemes based on the DLP or integer factorization.

5.5 Key Exchange Protocols for Elliptic Curve Cryptography

Many schemes have been proposed for secret sharing. Some of the popular key exchange schemes are Diffie-Hellman and the ElGamal protocol. This section describes the Diffie-Hellman and ElGamal protocol.

5.5.1 Elliptic Curve Diffie-Hellman Protocol (EC-DH)

Diffie-Hellman key exchange has been discussed in a previous chapter (Sec. 3.6.1). The EC key exchange is similar to that one expect for the fact that elliptic curve scalar multiplication is used as the one-way function instead of exponentiation.

The Elliptic curve Diffie-Hellman key exchange protocol works as follows.
Assume Alice and Bob are the two people in the communication and they want to generate a secret key. Both agree to use a specific curve, field size and representation of elements in the field. They also select a base point B on the curve. All these parameters are made public.

1. Alice selects a random value k_A
2. Alice computes $k_A B$.
3. Alice sends $k_A B$ to Bob.
4. Bob selects a random value k_B.
5. Bob then computes $k_B B$ and sends it to Alice.
6. Alice then computes $k_A (k_B B)$.
7. Bob computes $k_B (k_A B)$, which is the shared secret.

The strength of the EC Diffie-Hellman key exchange protocol lies in the fact that given B, $k_A B$ and $k_B B$ it is difficult to compute $k_B k_A B$. This is because to compute $k_B k_A B$ the adversary would have to compute either k_B or k_A from $k_B B$ and $k_A B$ respectively, which requires solving the ECDLP. The EC Diffie-Hellman key exchange protocol too has no known sub-exponential solution for a well-chosen set of parameters [FL00]. These parameters include selecting a non super singular curve with prime order, also the order of the base point B of the curve should be prime. The same level of security can be achieved using EC Diffie-Hellman key exchange protocol utilizing lesser number of key-

bits. For example, a 163-bit key elliptic curve technology offers the equivalent security strength of a 1024-bit RSA system [LD00].

5.5.2 Elliptic Curve ElGamal Protocol

The elliptic curve ElGamal protocol is a very useful protocol for key exchange, digital signature and message encryption. The steps involved in this protocol are as follows –

Assume Alice and Bob are the two people in the communication and they want to generate a secret key. Both agree to use a specific curve, field size and representation of elements in the field. They also select a base point B on the curve. All these parameters are made public.

1. Alice selects a random value k_A
2. Alice computes $P_A = k_A B$.
3. Alice sends P_A to Bob.
4. Bob selects a random value k_B.
5. Bob then computes $P_B = k_B B$ and sends it to Alice.
6. Suppose that P_m is the message point embedded on the curve and Alice wants to send this to Bob.
7. Alice chooses a random bit pattern r, and computes two points
   $P_r = r B$ and
   $P_h = P_m + r P_B$.
8. Alice then sends both P_r and P_h to Bob.
9. Bob computes $P_s = k_B P_r$ and subtracts this from P_h
   $P_m = P_h - P_s$

5.6 Elliptic Curve Digital Signature Algorithm (ECDSA)

The Elliptic Curve Digital Signature Algorithm is based on the EC – ElGamal protocol. This ECDSA is composed of 3 parts – ECDSA key generation, ECDSA signature generation and ECDSA signature verification. The following steps illustrate this algorithm.

Alice and Bob agree to use a specific curve E, field size and representation of elements in the field. They also select a base point B on the curve of order n. All these parameters are made public.

ECDSA Key Generation

1. Alice selects a random number k_A from the interval [1, n-1]
2. Alice then computes $P_A = k_A B$.
   Alice's private key is k_A
   Alice's public key is (E, B, P_A).

ECDSA Signature Generation (to sign a message m)

1. Alice chooses a random k from the interval [2, n-1].
2. Alice computes $(x_1, y_1) = B * k$.
3. Alice computes $r = x_1 \bmod n$ (if $r = 0$ then go to step 1).
4. Alice computes $k^{-1} \bmod n$.
   and $s = k^{-1} (SHA (m) + ar) \bmod n$
   (if $s = 0$, go to step 1)
   Signature for $m = (r, s)$.
5. Alice sends (r, s) to Bob.

ECDSA Signature Verification:

1. Bob receives (r, s) and verifies that they are integers in the interval [1, n-1].
2. Bob computes $w = s^{-1} \bmod n$ and SHA (m).
3. Bob computes $u_1 = SHA (m) w \bmod n$ and
   $u_2 = r w \bmod n$.
4. Bob computes $u_1 B + u_2 P_A = (x_0, y_0)$
   and $v = x_0 \bmod n$
5. Bob accepts signature if and only if $v = r$.

5.7 Embedding Raw Data on the Curve

In order to transmit a message using elliptic curve, the message has to be embedded on the curve as a point. The message can then be encrypted and sent to the recipient. The trick is to hide the plaintext on the curve along with some random bits. The maximum number of bits that can be embedded on the curve is the field GF (2^m) is m-4 [ROS99]. The plaintext is converted into binary format using some well-known algorithms. Plain text can be represented in ASCII, which can then be converted into binary using ASCII to binary conversion methods. Or a set of numbers could be assigned to the text, which is known to both parties. This binary message is then appended with some random bits, the values of which can be changed to satisfy the curve equation. This is the x coordinate of the message point. The equation $y^2 + xy = x^3 + ax + b$ is then solved to get the y coordinate of the message point. Then various protocols discussed above can be applied to hide this point on the curve.

There exist only probabilistic methods for embedding plaintext on an elliptic curve. There is no deterministic polynomial time algorithm for the same [ROS99].

Chapter 6

Algorithms used in computation of the elliptic curve scalar multiplication

As seen in fig 5.6, the elliptic curve scalar multiplication is a series of point doublings and point additions. Various algorithms have been proposed for point multiplication which are discussed later in this chapter. These algorithms are divided into two important cases i.e., generic point multiplication algorithms and fixed point algorithms. In the generic case, multiplication is done using arbitrary points on the elliptic curve whereas the point is fixed in fixed point multiplication [OP00a]. Fixed point multiplication algorithms are more efficient and faster than generic algorithms; however, they require more precomputations and hence have more storage requirements. Some other factors that influence speed of these algorithms include coordinate selection and selection of the irreducible polynomial [BDT02].

6.1 Irreducible Polynomial

A polynomial which cannot be factored into smaller degree polynomials in the same field is an irreducible polynomial. The degree of the irreducible polynomial is equal to "m" for the field GF (2^m). Multiplication and exponentiation operations in the finite field are carried out modulo the irreducible polynomial. [IEE98, ANS99] standards recommend the use of trinomials of the form $x^m + x^t + 1$ or pentanomials ($x^m + x^a + x^b + x^c + 1$) as the irreducible polynomial because they greatly simplify implementations. Trinomials ($x^m + x^t + 1$) as seen from the equation are polynomials with only 3 non-zero coefficients and pentanomials have 5 non-zero coefficients. For elliptic curve arithmetic in the field GF (2^m), trinomials are of the form $x^m + x^t + 1$ where $0 < t < m$. Pentanomials are of the form $x^m + x^a + x^b + x^c + 1$ where $0 < a, b, c < m$ and $a \neq b \neq c$. An arbitrary irreducible polynomial can contain a maximum of m+1 non-zero coefficients. [OP00a] gives an approximate bit-complexity for using different irreducible polynomials for GF (2^m) arithmetic operations, assuming the number of non-zero coefficients in the arbitrary polynomial is r + 1.

$GF(2^m)$ operation	Irreducible Polynomial		
	Arbitrary	**Trinomial**	**Pentanomial**
Addition	m	m	m
Square	rm	2m	4m
Multiplication	$2m^2 + (r-2)\,m - r + 1$	$2m^2 - 1$	$2m^2 + 2m - 3$

Table 6.1: Irreducible Polynomial bit complexity for arithmetic operations [OP00a]

The work described here uses trinomials or pentanomials depending on the field, for implementation because it greatly simplifies implementation and enhances speed of algorithms.

6.2 Coordinate representation

Another important factor that influences the speed of implementation is the coordinate representation of the elliptic curve points. Using projective coordinates (X, Y, Z) instead of affine coordinates (x, y), eliminates the need for inversion in the scalar multiplication algorithm, which is the most expensive operation in terms of time and resources required [BDT02].

6.2.1 Projective Space

Definition: "The projective space over a field F is the set of equivalence classes of tuples $(X_0, X_1 \ldots X_n)$ (not all components zero) where two tuples are said to be equivalent if they are scalar multiplies of one another" [OSW02]. Such an equivalence class is called a projective point, which means that a point (X, Y, Z) is the same point as (aX, aY, aZ) where $a \neq 0$.A point in the projective plane has three coordinates (X, Y, Z). The conversion of affine (x, y) to projective is as follows:

$X = x, Y = y, Z = 1.$

And Projective can be converted back to affine as

$x = X/Z, y = Y/Z$

Substituting the above values of x and y in the nonsupersingular elliptic curve equation 5.1 we get the corresponding equation for an elliptic curve in projective coordinates:

$Y^2 Z = X^3 + a X Z^2 + b Z^3.$

6.2.2 Elliptic Curve Operations using Projective Coordinates

The following shows the mathematical formulae for elliptic curve point addition and point doubling operations in projective coordinates.

Point Addition

Let P (X_0, Y_0, Z_0) and Q (X_1, Y_1, Z_1), $P \neq -Q$ be the two points to be added and let $R (X_2, Y_2, Z_2) = P + Q$.

Then,

Let $V = (X_0 Z_1^2 + X_1 Z_0^2)(X_0 Z_1^2 - X_1 Z_0^2)^2 - 2 X_2$

$X_2 = (Y_0 Z_1^3 - Y_1 Z_0^3)^2 - (X_0 Z_1^2 + X_1 Z_0^2)(X_0 Z_1^2 - X_1 Z_0^2)^2$

$2 Y_2 = V (Y_0 Z_1^3 - Y_1 Z_0^3) - (Y_0 Z_1^3 - Y_1 Z_0^3)(X_0 Z_1^2 - X_1 Z_0^2)^3$

$Z_2 = Z_0 Z_1 (X_0 Z_1^2 - X_1 Z_0^2)$

Point Doubling

Let P (X_1, Y_1, Z_1), R $(X_2, Y_2, Z_2) = 2$ P.

$X_2 = (3 X_1^2 + a Z_1^4)^2 - 8 X_1 Y_1^2$

$Y_2 = (3 X_1^2 + a Z_1^4)(4 X_1 Y_1^2 - X_2) - 8 Y_1^4$

$Z_2 = 2 Y_1 Z_1$

It can be seen from the above that point addition and point doubling using projective coordinates do not require an inverse operation.

The table below shows a comparison of the number of operations required to perform a point addition and double in affine and projective coordinates for the field GF (2^m).

Operation	Affine	Projective
Point addition	1 inv + 2 mul + 1 sq	15mul + 5 sq
Point doubling	1 inv + 2 mul + 1 sq	5 mul + 5 sq

mul – multiplications

sq – squares

inv – inverse

Table 6.2: Number of Operations in affine and projective coordinates

The following table shows the bit complexities of GF (2^m) arithmetic operations

GF (2^m) operation	Bit complexity
Addition	O(m)
Square	O(rm)
Multiplication	$O(m^2)$
Inverse	$O(m^2 \log_2 m)$

Table 6.3: Complexity of GF (2^m) arithmetic operations

It can be seen from table 6.3 that inverse is the most complex operation. m is in the order of 163 to 571 for cryptographic applications [OP00a]. Therefore, it is beneficial to use projective coordinates which completely eliminate the need for inversion. Multiplication in binary fields can be made faster by using different multiplication algorithms discussed later. Using a digit multiplier, the complexity of multiplication is O (m/d), d is the digit size, which is faster than the O (m^2) multiplier.

6.3 Point Multiplication Algorithms

The most common algorithm for scalar or point multiplication is the double and add algorithm which has also been documented in the IEEE 1363 draft for public key cryptography standards. In this algorithm every bit of k is scanned starting from the MSB and ending at the LSB. If the bit is set (1), then a point double operation is performed. If the bit is 0, a point addition is performed along with the point double. The double and add method is illustrated in the pseudo-code below.

Input $k = (k_m, k_{m-1}.....k_1, k_0)$, P, where $k_i \in 0, 1$
Initialize $Q = 0$
for $i = len(k)- 1$ to $i = 0$ **loop**
    if $k_i = 0$ then
        $Q = Q + P$ *//if bit is 0, the perform point addition*
    end if;
        $Q = 2Q$ *//point double for each bit of k*
end loop;

Output Q = kP

If half the bits are set or 1 in the binary representation of k (k is m –bit), then the double and add algorithm requires 'm' point doublings and 'm/2' point additions. The number of operations can be minimized to m/3 additions by using addition-subtraction chains, keeping the number of point doublings fixed [BDG02]. The addition-subtraction chain method is also known as the Non Adjacent Form (NAF) method and is based on the fact the computing the additive inverse of P, i.e., -P is very simple as –P is the reflection of P along the X-axis (Sec. 5.1).

The NAF method for point multiplication is faster than the double and add method. k is represented in NAF or signed binary representation, with smaller number of non-zero digits. NAF is unique for every number. The algorithm for the NAF method is shown below.

Input $k = (k_m, k_{m-1}.....k_1, k_0)$, P, where ki ϵ 0,1,-1
Initialize Q = 0
for i = len(k)- 1 to i = 0 **loop**
    if $k_i = 0$ **then**
        $Q = Q \pm k_i P$ //if bit is 0, the perform point addition
    end if;
        Q = 2Q //point double for each bit
end loop;
Output Q = kP

Other methods were proposed by [AMV93, LKP03], where instead of using a random k, the bits of k were selected such that k had a small Hamming weight (HW). Hamming weight of a number is the number of non-zero bits in its binary representation. By choosing k in this manner the number of point additions could be minimized.

The only difference between these two methods was that k in the [LKP03] method was represented in signed binary representation, so that there were more possible choices for k in a given number of bits. Both methods required m-1 doublings and w-1 additions/subtractions for a point multiplication operation (w being the HW of k).

Method	Point Doubling	Point Addition
Double-and-Add	m	m/2
NAF method	m	m/3
[AMV93, LKP03]	m-1	w-1

Table 6.4: Comparison of point multiplication algorithms

Table 6.4 shows a comparison of the point multiplication algorithms discussed in terms of number of point doubling and point addition operations needed in each.
In fixed point algorithms, 2^kP is precomputed for different values of k and stored, since P is fixed and known. These methods are faster than the generic algorithms discussed above, but have more storage requirements for the precomputed values [OP00a].

6.4 Montgomery Point Multiplication Algorithm for GF (2^m)

This algorithm was introduced by [LD99] and it is an optimized version of the original Montgomery method [OM98] for scalar multiplication in binary field. It is one of the most efficient methods, requires no precomputations and is faster than other known methods [LD99]. One of the fastest reported implementations of elliptic curve processor [OP00a] also uses this algorithm for point multiplication. The algorithm has been proposed for both affine and projective coordinates. The projective coordinate version is much more efficient [LD99].

The method starts by defining 3 points P_1, P_2 and P such that $P_2 = 2P$ and $P_1 = P$.
P is the difference $P_2 - P_1$.
Montgomery's scalar multiplication algorithm illustrated in the following pseudo code.

Input – P, k

Initialize - $P_1 = P$ and $P_2 = 2P$

for i in len(k)-2 downto 0 **loop**

if $k_i = 1$

$P_1 = P_1 + P_2$

$P_2 = 2 P_2$ // $P = P_2 - P_1 = 2 P_2 - P_1 + P_2$

else

$P_2 = P_1 + P_2$

$P_1 = 2 P_1$ // $P = P_2 - P_1 = P_1 + P_2 - 2 P_1$

end if

end loop;

Output $P_1 = kP$

At each step of the iteration the sum of two points P_2 and P_1 is computed using the x – coordinates of the points only and the difference between the points is maintained constant. The y coordinate is obtained at the end of the point multiplication process.

In the projective coordinate version of the algorithm the x coordinate is converted into projective using X and Z in projective space as $x = X/Z$ for both points P_1 and P_2. The y coordinate is not used in this algorithm. The resulting coordinates of the point multiplication process are converted back into affine.
The following is the pseudo code for the Montgomery scalar multiplication algorithm.

Pseudo Code for Elliptic Curve Scalar Multiplication

Inputs:
Elliptic Curve Parameters – Irreducible Polynomial, a, b
Coordinates of point P x, y
Random key – k

1. Convert (x,y) to projective coordinates, (X_1,Z_1,X_2,Z_2) = affinetoProjective(x,y)
2. for I = length (k)-2 downto 0
   if k (i) = 1 then

(X_1, Z_1) = madd (X_1,Z_1, X_2, Z_2,x)
(X_2,Z_2) = mdouble (X_2, Z_2)

else

(X_2,Z_2) = madd (X_1,Z_1,X_2,Z_2, x)
(X_1, Z_1) = mdouble (X_1, Z_1)

end if;
end for;

3. Convert back to affine (x, y) = projtoaffine $(X_1, Z_1, X_2, Z_2, x, y)$

Output (x, y) = k P

Functions:

*/*This function converts the x coordinate from affine to projective. The y coordinate is not used in this algorithm */*

affinetoProjective(x, y)

$X_1 = x$;
$Z_1 = 1$
$X_2 = x^4 + b$;
$Z_2 = x^2$
return (X_1,Z_1,X_2,Z_2)

*/*This function performs the point addition operation in projective coordinates*/*

madd(X_1,Z_1,X_2,Z_2, x)
$X_1 = X_1Z_2X_2Z_1 + x\,(X1Z2 + X2Z1)^2$
$Z_1 = (X_1Z_2 + X_2Z_1)^2$
return(X_1,Z_1)

*/*This function performs the point doubling operation in projective coordinates*/*

mdouble(X,Z)
$X = X^4 + bZ^4$
$Y = X^2Z^2$

*/*This function converts the projective coordinates back into affine*/*

projectivetoaffine (X_1,Z_1,X_2,Z_2,x,y)
$xk = X_1/Z_1$;
$yk = ((X_1/Z_1 + x)(X_2/Z_2 + x) + x^2 + y) * (X_1/Z_1 + x)/x + y)$

return(xk,yk)

The algorithm starts by converting the x coordinates of P_1 and P_2 into projective form using function affinetoProjective (x, y). The relation between P_1 and P_2 (P_1- P_2 = P) is maintained at each step of the iteration. If $k_i = 1$; P_1 is set to P_1+ P_2 and P_2 is set to 2 P_2. Similarly when k_i is 0, then P_2 is set to P_1+ P_2 and P_1 is set to 2 P_1. P_1 contains the value of kP at the end of the iteration, which is converted back to affine coordinates at the end of the iteration process. As can be seen from the pseudo code there is no inversion operation; except in a last step of coordinate conversion (projectivetoaffine ()).

This method has less storage requirements and computational complexity. The following table lists the number of multiplications, squares and inversions required in the scalar multiplication process using Montgomery's algorithm.

Multiplications	Squares	Inverses
6m + 10	5m + 3	1

Table 6.5: Computational complexity of Montgomery point multiplication algorithm [OP00a]

6.5 Inversion Algorithm

Some of the most commonly used algorithms for inversion in finite binary fields are the Extended Euclidean Algorithm [MOV96], Fermat's Little Theorem [OP00b], Look up Table [PAR95] method and the Itoh-Tsujii [IT88] algorithm.

The Euclidean algorithm is used for calculating the greatest common divisor (GCD) of two numbers. It states that the GCD of two numbers can be expressed as a linear combination of the two numbers. If F(x) is the irreducible polynomial in the field GF (2^n), then this algorithm states that GCD (F(x), A(x)) = 1 (A is an element of GF (2^n)) and there exists a polynomial B(x) which for some C(x) satisfies the equation

$A(x)\,B(x) + C(x)\,F(x) = 1$

i.e., $A(x)\,B(x) = 1 \bmod F(x)$.

B(x) is the inverse of A(x) in GF (2^n).

This computation is based on repeated application of the division-mod algorithm and therefore the Extended Euclidean algorithm can also be used to find the inverse of an element in finite field.

A simpler method is the look up table based method where the inverses and field elements are precomputed and stored. This requires a lot of storage and because fields are large for cryptographic applications, it is used generally for subfield inversion and finds use in block ciphers [PAR95].

The Itoh-Tsujii method is very popular for use in binary finite fields with normal basis representation. This is because it is an exponentiation based method and exponentiation can be computed fairly easily in normal basis. Squaring in normal basis is simply a cyclic shift of coefficients. [GP02] proposed a method for adapting the Itoh-Tsujii algorithm for polynomial basis. However this is efficient for only certain classes of finite field because complexity of exponentiation is lower in these classes.
The elliptic curve processor prototyped in this work used the Fermat's Little Theorem for computing inverse. This method is not restricted to a particular form of extension field and is a simple algorithm, which is easy to implement and takes up less code space [GBK01]. It is also an exponentiation-based method and does not require any precomputataions. Fermat's theorem requires [$\log_2$ (n-1) + HW (n-1) -1] multiplications and k-1 squarings in GF (2^n) [PAR95].

6.5.1 Fermat's Little Theorem

The theorem states that for any A$\in$ GF (2^m), the inverse is computed as follows
$A^{-1} = A^{(2\hat{}m - 2)}$

The exponentiation is computed using the **left-to-right binary exponentiation** method.
If L is the length of the binary representation of the exponent then left-to-right binary exponentiation algorithm performs L squarings and N – 1multiplications with the number where N is the number of 1's in the binary representation of the exponent. The number of multiplications can be reduced by forming addition chains (a sequence of pairs which

satisfy certain mathematical properties and reduce multiplication to addition), but determining the shortest addition chain is a NP- hard mathematical problem [MOV96].

Chapter 7

This chapter describes the entire design and implementation cycle. Also the tools, language and methodology used in the implementation are discussed. An introduction to the target FPGA, its architecture, advantages and disadvantages are also presented here.

Design and Implementation Steps

1. Define design specification/ goals
2. Define design in VHDL
3. Simulate the source code (Design verification)
4. If simulation works, synthesize, optimize, place and route and download on target FPGA
5. Test Design

All the steps are interrelated and are not in any definite order. Each step needs to be retraced at some point in the process, to ensure correct implementation.

7.1 Methodology

The design of the ECP was developed in VHDL and placed on an FPGA. Functional simulation of the various blocks was carried out to ensure proper functionality. The ECP was tested using various test vectors. The test vectors for the coordinates of the point were generated using a random number generator coded in Java. Another Java code was written to generate the elliptic curve parameters. The instruction set for the algorithm used in the point multiplication process is loaded into the instruction memory of the target FPGA. The instruction set is specified in a ".mif" file for Altera and a ".coe" file for Xilinx FPGA. The instruction memory is configured to read the instructions from the files. Both the instruction memory and data memory are configured as dual port RAMs (DPRAM). They can therefore be accessed by the host machine through 32-bit/66 Mhz PCI interface [XILINX]. Also the DPRAM configuration allows the host processor to load different programs without having to reconfigure the elliptic curve processor.

7.2 Overview of VHDL

The elliptic curve processor architecture was developed using Very High Speed Integrated Circuit Hardware Description Language (VHDL). VHDL is a programming language for digital hardware. VHDL was adopted as a standard in 1987 [WSC89].

There are many advantages of using VHDL for modeling digital hardware. One of the biggest advantages is that VHDL allows reusability of design. Various components like flip flops, registers, memory etc can be used in many different designs. Model libraries can be made for most commonly used function blocks. VHDL is also accepted as a standard and has both Government and Industry support.

VHDL code is portable. The code developed for this thesis was compiled and simulated for both Altera and Xilinx platforms. VHDL is also foundry and technology independent which makes it easier for the designer to code without having to wait for selecting a particular technology. Another important aspect of VHDL is that the design can be built using modules, which can be later integrated into a complete system. This simplifies the design process. Also it helps to debug and functionally simulate each unit separately.

7.3 CAD Tools

Two CAD tools were used for the two target FPGAs available. Typical design flow through a CAD tools is as follows –

1. Design Entry
2. Translation
3. Optimization and Synthesis
4. Device Fitting
5. Simulation
6. Device Programming

7.4 Target FPGA

Two different FPGAs were used to prototype the elliptic curve processor. The Xilinx FPGA has more number of gates, dual ported memory and incorporate fast carry logic. The Altera FPGA has dual port memory, lesser number of gates and was used to compare the performance of the elliptic curve processor on two different platforms. Some of the features of both the FPGAs are highlighted below.

Altera Flex10K70

The Flex device is a SRAM-based FPGA. It has 70,000 gates with 3,744 logic elements (LE), 468 logic array blocks (LAB), 9 embedded array blocks (EAB) and 18,432 RAM bits. The Flex device is built on a UP2 education board, which also includes a MAX 7000 (CPLD) device. There are 240 I/O pins on the board.

Xilinx Virtex XCV1000bg560

The Virtex device has 27, 648 logic cells, 1,124,022 system gates and 3 speed grades. This is also a SRAM-based FPGA. It has configurable logic blocks (CLB), which are similar to the LABs in the Flex device. The board provides 404 I/O pins.

7.5 Platform Options

The figure below shows the various available platform options for implementing the elliptic curve processor.

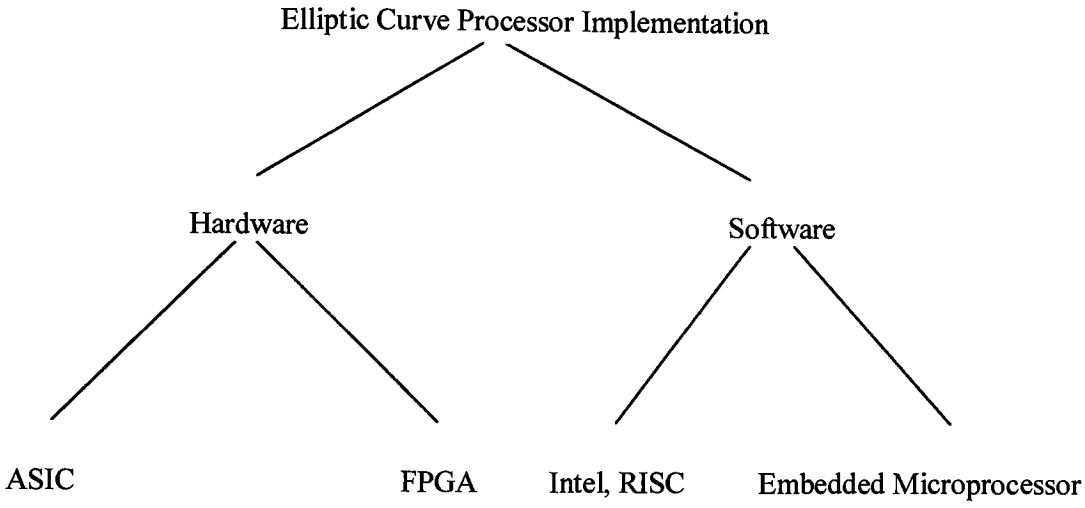


Fig 7.1: Platform Options

Elliptic curve cryptosystems have been successfully implemented on most of these platforms [Sec 2.1, 2.2].

Some of the desired characteristic of the elliptic curve processor are - space effective, power efficient, cost effective, small design and development cycle and flexible, which lends itself easily to customization. This is necessary because the design of the elliptic curve cryptosystem is targeted for use in constrained environments such as mobile devices and smart cards.

The table below summarizes the features of each implementation platform.

	Performance	Cost	Power	Flexibility	Design Effort
ASIC	High	High	Low	Low	High
Processor	Medium/Low	Medium/Low	Medium	High/Medium	Medium/Low
Reconfigurable Hardware	High/Medium	Medium/Low	High/Medium	High/Medium	Medium/Low

Table 7.1: Features of Implementation Platforms

ASIC-based implementations provide very high performance, consume less power and have a smaller chip count for the overall design. However designing in ASIC requires considerable effort, long fabrication time which leads to longer design cycles. They are also not flexible and are usually very expensive [EYC00].

Software based applications are flexible and do not take as long to develop. They provide medium performance level and have medium cost. The instructions in any software language are execute sequentially so parallelism cannot be well exploited. Also the hardware used is not specialized for a specific function. They also consume high power.

7.5.1 Reconfigurable Hardware

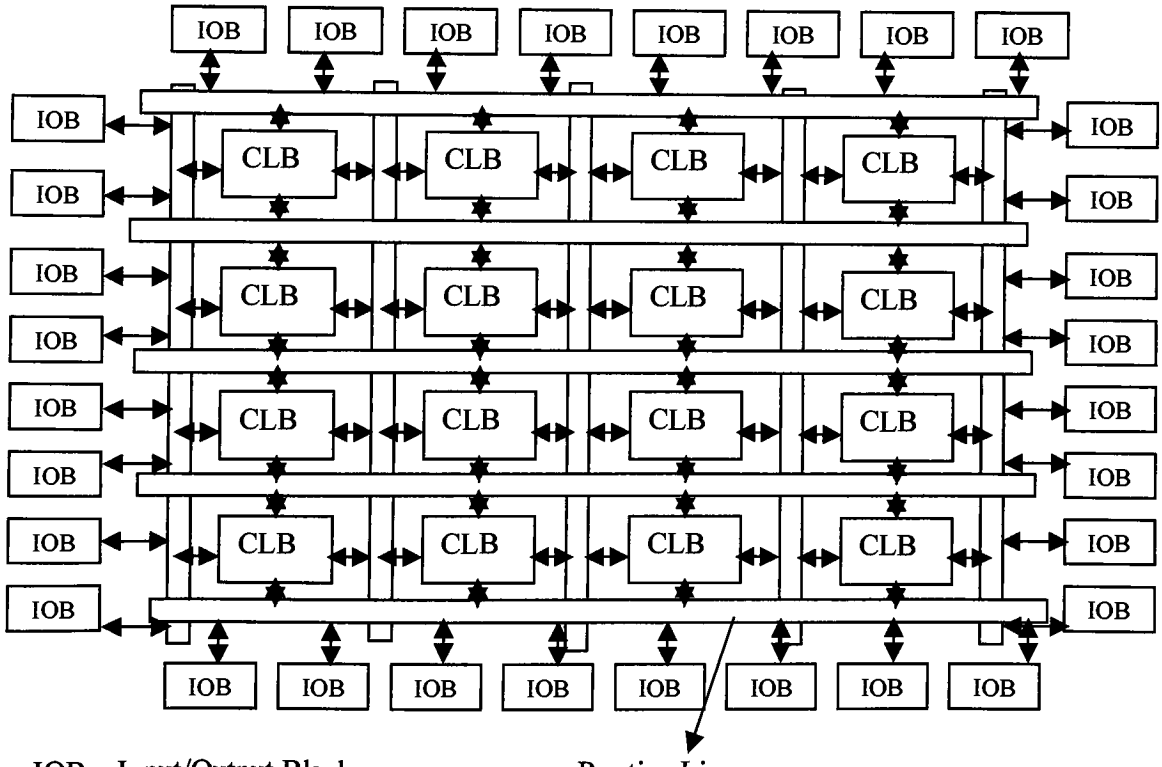
Reconfigurable hardware includes devices in which the functionality of logic gates is customizable at runtime. It is a general-purpose hardware configured for a specialized function. Implementation on the reconfigurable platform is highly flexible in the sense that the algorithms and system parameters can be changed at any time by reconfiguration. It gives reasonable performance with low cost compared to ASIC implementations [EYC00]. ASIC lose the flexibility to adapt to circuit changes once manufactured. ASICs are not manufactured in large volumes because they have rigid fabrication rules, such as 0.35 or 0.5 micron, whereas FPGAs are manufactured in large quantities as they can easily be migrated to the best available fabrication technology and are therefore cost effective [XILINX]. The reconfigurable hardware approach combines the speed and concurrency of hardware with the flexibility of software. Therefore this is a good platform for implementing cryptographic applications.

7.6 Introduction to FPGA

Field Programmable gate array (FPGA) is an integrated circuit whose internal functional operation is defined by the user. The user specifies the function after the device has been manufactured, and all function changes can take place at the user's site.

An FPGA contains several copies of programmable logic element or logic block. These logic blocks are used to implement digital logic i.e., logic gates. Logic elements are connected to one another on the chip using a programmable interconnection network. Maximum clock rates that can be obtained using FPGA devices are in the range of 50 MHz to 200 MHz [XILINX].

The logic blocks in a FPGA are organized in an array with interconnection network between them.



IOB – Input/Output Block

CLB – Configurable Logic Block

Fig 7.2: Basic Structure of a Xilinx FPGA

There are basically two types of FPGA

1. SRAM (Static Random Access Memory)-based FPGA
2. One-time Programmable FPGA.

SRAM-based FPGAs use SRAM to define the logic and SRAM-controlled interconnect switches whereas One-time Programmable FPGA use anti-fuse switches.

7.6.1 Configurable Logic Blocks (CLB) or Logic Array Blocks (LAB)

These form the heart of the FPGA device. They contain the programmable logic for designing different logic functions. These blocks are also used as RAM/ROM blocks or registers for storage. They provide dedicated capability for high-speed arithmetic functions. They can also serve as routing elements because they have multiplexers to route logic within the block and to and from external resources.

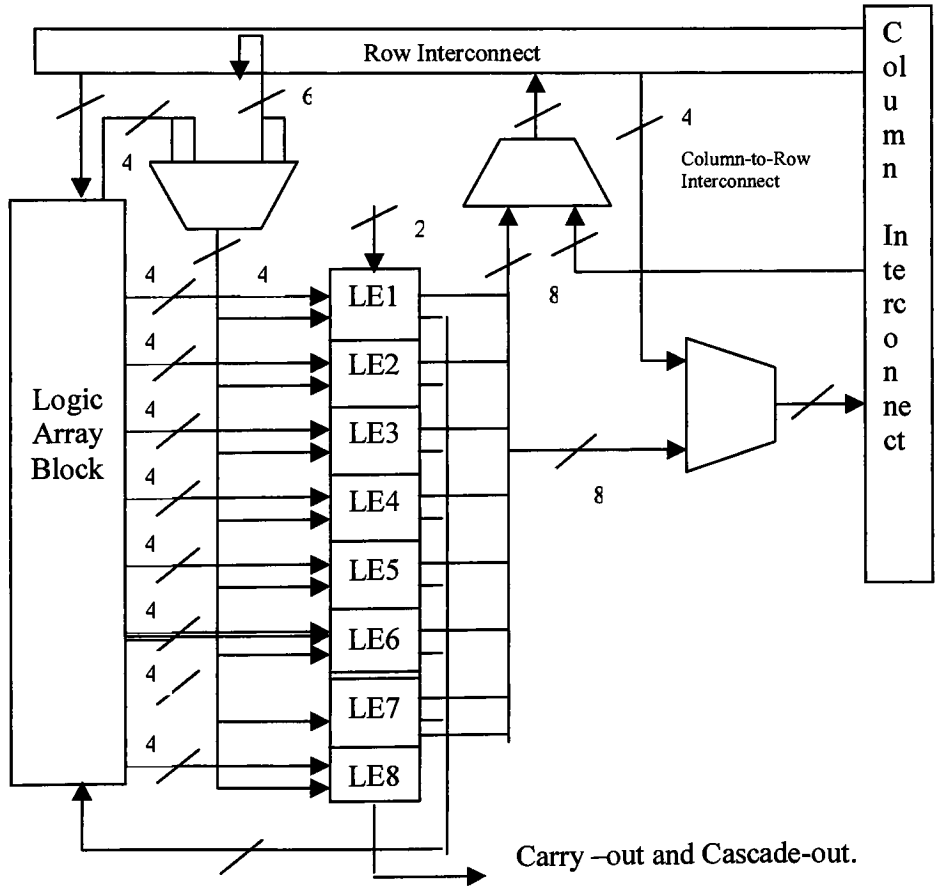


Fig 7.3: Flex 10K Logic Array Block

Altera devices have logic array blocks (LABs) which are macrocells combined into groups of 16 [ALTERA]. The architecture of a logic array block is shown in the figure above. The programmable interconnect routes data to or from LABs or external pins on the device.

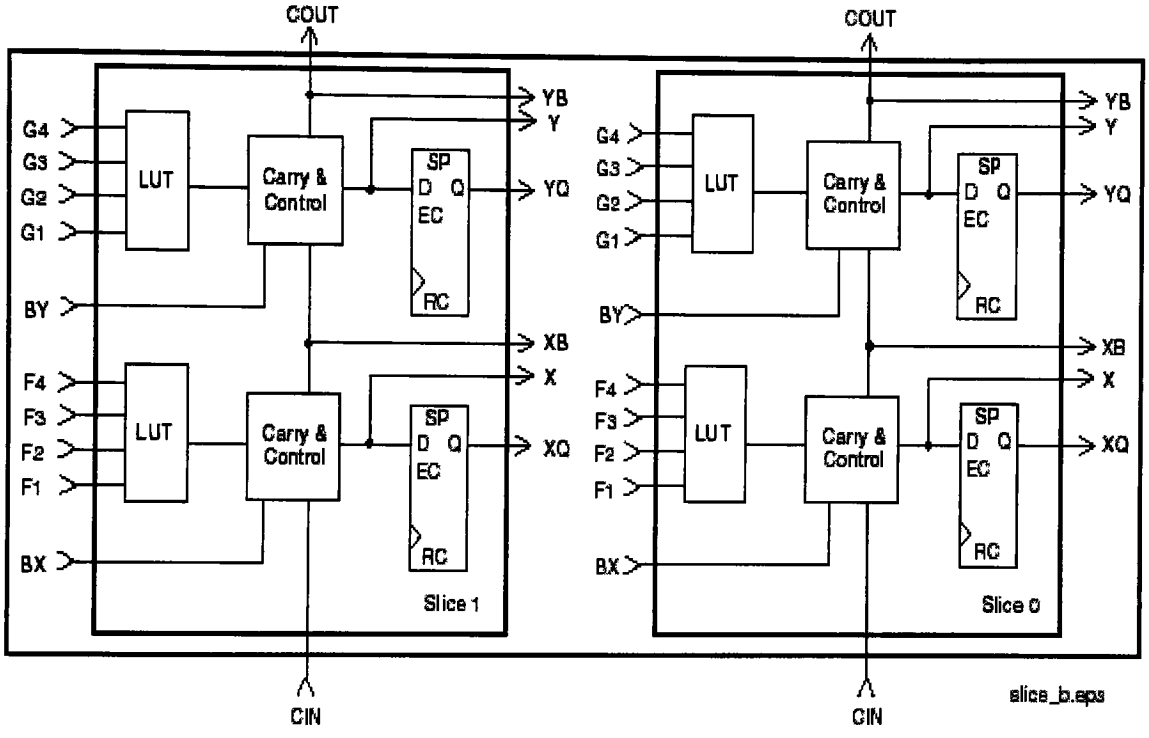


Fig 7.4: 2-Slice Virtex CLB [XILINX]

Xilinx devices have a more complex logic element called CLB. Each CLB includes 4, 4-input loop-up table (LUT), carry logic and a storage element. Outputs from the LUTs can be fed into flip flops and routed to other CLBs.

7.6.2 Input/Output Blocks (IOB)

The Input/Output blocks provide the interface between the package pins and the configurable logic blocks (CLB). These transmit the signals from the chip and the host. Each block can be used either as an input or output or a bidirectional signal. The input buffer accepts different threshold levels, typically CMOS and TTL, so that the FPGA can interface to these devices. Similarly the output buffer is configurable so that the FPGA can output the most standard TTL or CMOS levels. All the IOBs not used in the configuration are set to high-impedance state in order to isolate the internal logic.

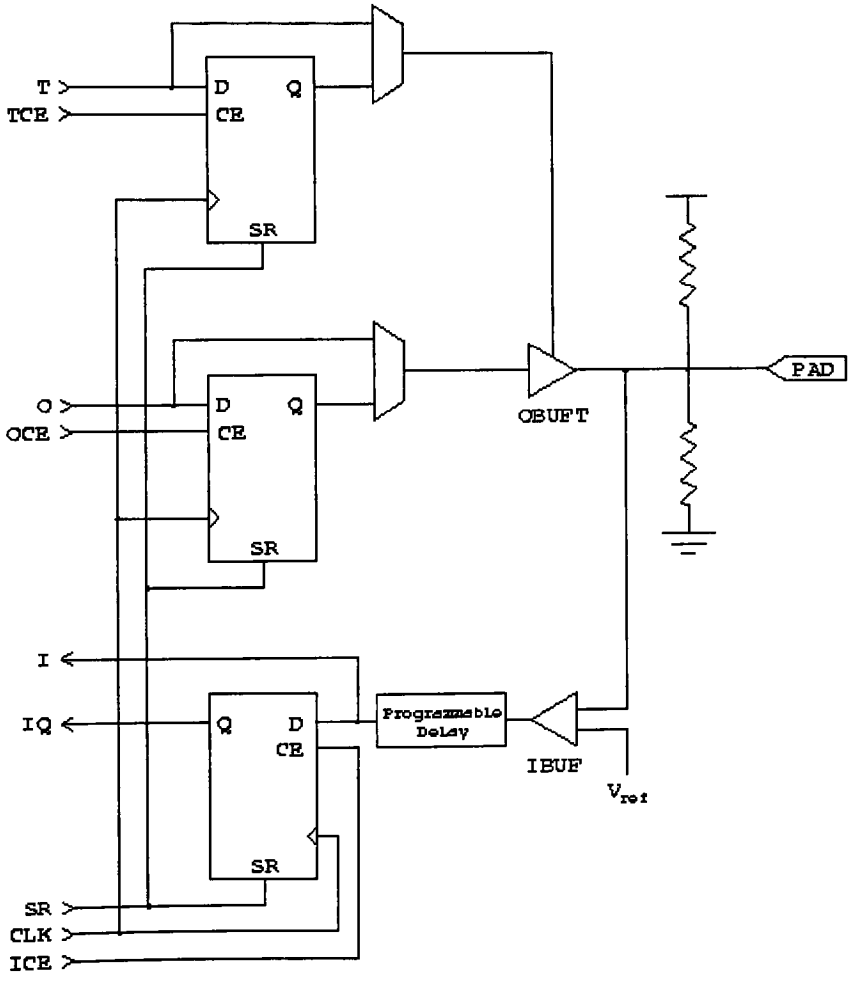
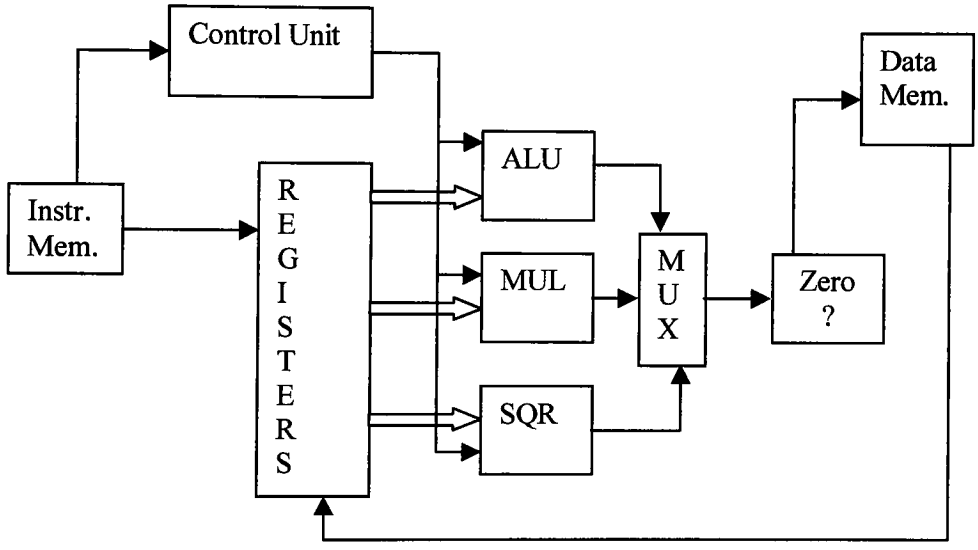


Fig 7.5: IO block [XILINX]

Chapter 8

Elliptic Curve Processor (ECP)

The architecture proposed for the elliptic curve processor is a RISC (Reduced Instruction Set Computer) – based architecture, prototyped on an FPGA.
The elliptic curve processor can be easily reprogrammed to use different algorithms by changing the instructions. The arithmetic unit of the processor is designed to support finite field arithmetic.



Instr. Mem. – Instruction Memory
Data Mem. – Data Memory

Fig 8.1: High-level view of the ECP

The proposed processor architecture can be adapted to any kind of extension field and elliptic curve. Pipelining is introduced so as to maximize instruction throughput. The ECP processor is built on the MIPS (Microprocessor without Interlocked Pipeline Stages)

processor core for RISC-based systems. This is a specialized cryptographic processor and not a modified general purpose CPU. A complete description of the MIPS core can be found in [PH97]. The ECP is a low-power processor, which is an important consideration for using it in constrained environments. The method used to reduce power consumption in the ECP is termed as clock gating i.e., if a certain part of a processor is not needed for a given operation, then it does not require the clock signal. In absence of the clock signal there is no switching, and since switching requires power, its absence results in lower power consumption.

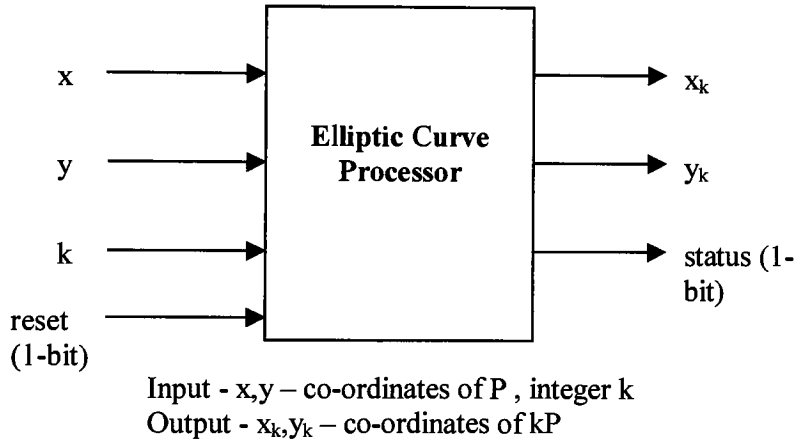


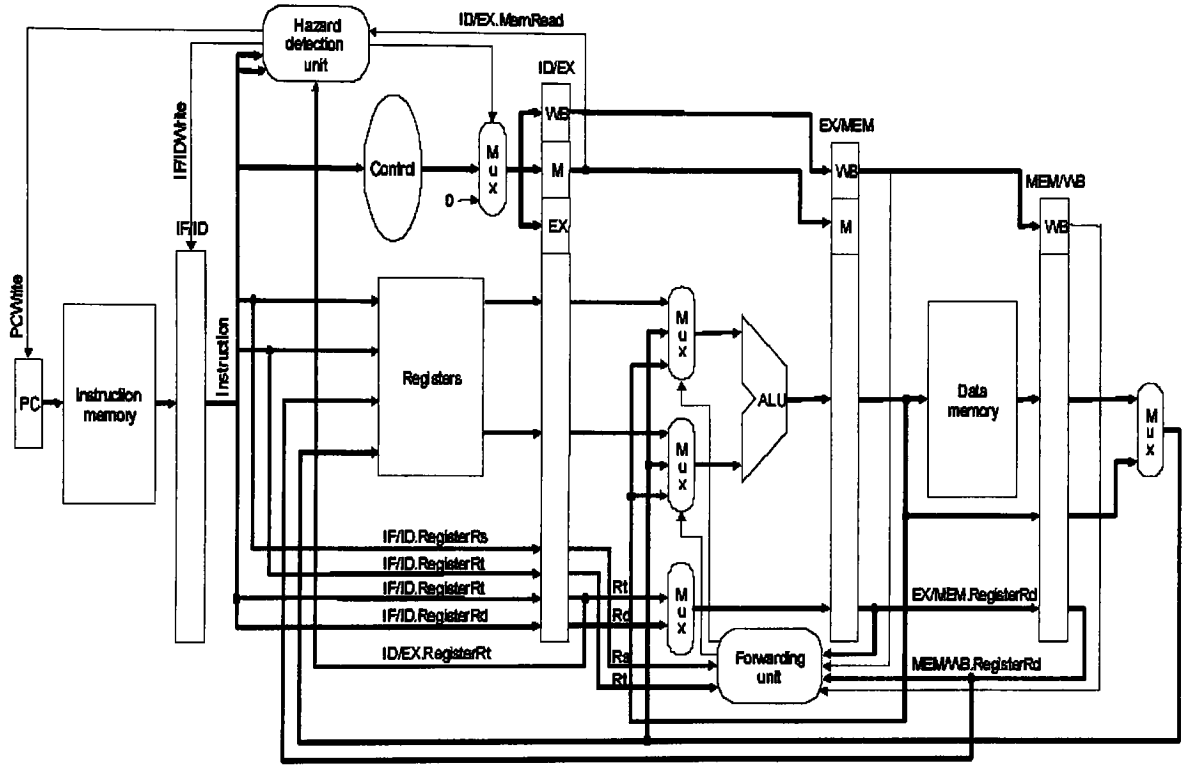
Fig 8.2: Basis Processor Operation

8.1 Basic Operation

The host enters the parameters for the scalar multiplication. Upon receiving the parameters, the ECP starts the processing using instructions from the instruction memory. Instructions are loaded into the instruction memory using the *.mif or the *.coe file The coordinates are first converted from affine to projective and then Montgomery's point multiplication algorithm discussed above is used for computing the point multiplication. The host can also input parameters for the next multiplication before the first one

completes. After completion of the multiplication, the ECP outputs a 'done' status. The host can now read the result of the multiplication operation from the ECP.

8.2 Architecture of Processor:



8.3: Block Diagram of the Processor showing Pipeliue Stages [PH97]

8.2.1 Datapath

The instruction and data memory can connect the Elliptic curve processor with the PCI (Peripheral Component Interconnect) bus of the host system, as these are dual ported RAM memory which have interfacing capability provided by both Altera and Xilinx. The data bus, register file and data memory is 163 bits wide. The address bus is 8-bit and the

instruction length is 32-bit. The instruction memory and data memory are built using standard macros available for the target FPGA. It has 22 (163-bit) general purpose registers.

The instructions are executed in a 5-stage datapath – instruction fetch, instruction decode, execute, memory and write-back.

8.2.2 Instruction Set of the Elliptic Curve Processor

The processor incorporates 32-bit instructions.

Format of Instructions

Field Size	6-bits	5-bits	5-bits	5-bits	5-bits	6-bits
R-format	Opcode	R_S	R_T	R_D	Shift	Function
I-format	Opcode	R_S	R_T	Address/immediate value		
J-format	Opcode	Branch Target Address				

R_S – register containing operand 1

R_T – register containing operand 2

R_D – register containing result

Table 8.1: Format of Instructions

As seen from the figure above, ECP has three instruction formats. The R-format instructions perform operations on data in the registers. These operate on two operands and the result is stored in the register. The I-format instructions are the load and store instructions which reference the memory. The J-format instructions include the branch and Jump instructions.

Instruction Set

Instruction	Syntax	Format
Load	LOAD R_S, R_T	I
Store	STORE R_S, R_T	I
Add (XOR)	ADD R_S, R_D , R_T	R
Multiply	MUL R_S, R_D , R_T	R

Square	SQ R_S, R_T	R
Inverse	INV R_S, R_T	R
Shift Left	SLL	R
Branch on equal	BEQ R_S, R_T	J
Jump	JMP	J

Table 8.2: Instruction Set

This limited instruction set is adequate to realize the Montgomery scalar multiplication algorithm. The complete assembly language code for computing the point multiplication is available in Appendix.

8.2.3 Control Unit

The control unit takes as input the instruction opcode and generates control signals for different units in the processor. Depending on the opcode, it decides the type of instruction. It also examines the instructions for hazards that may result from pipelining instructions and sends appropriate signals to the exception units to resolve the hazards and to the different pipelined registers in the processor. The control unit is designed using finite state machines.

8.2.4 Exception Units

The processor incorporates two units for handling exceptions that may occur in the normal flow of instructions. These units are the forwarding unit and the hazard detection unit.

Forwarding Unit

The forwarding unit, which is also known as register-bypassing is used to eliminate data hazard stalls. Using this hardware, an instruction needing a result from a previous instruction need not wait for the previous instruction to write data back to the register file. The forwarding unit forwards the result to the instruction needing it directly from where it was produced.

Hazard Detection Unit

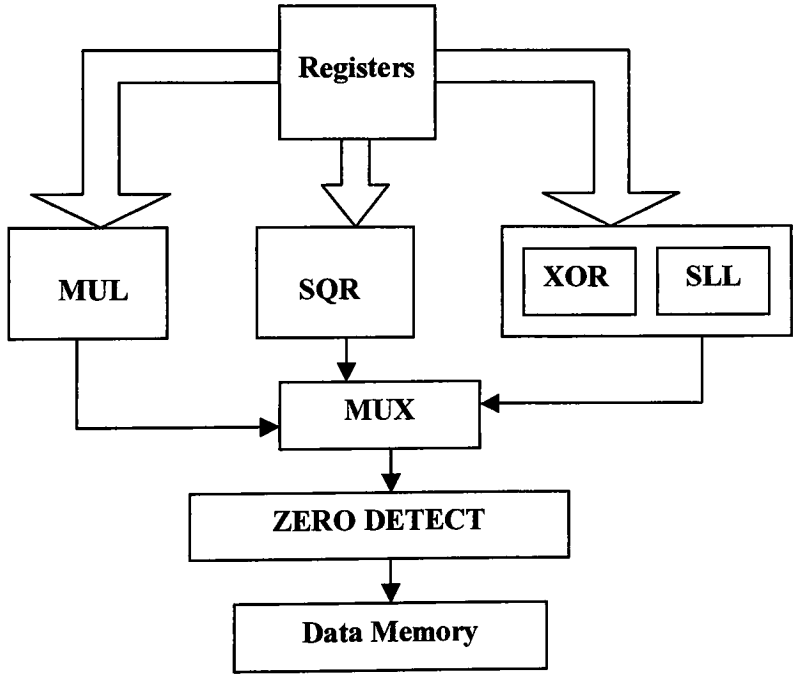
The hazard detection unit detects a load instruction followed by an instruction that used the loaded value and a stall cycle is introduced. The stall cycle is needed even with the forwarding unit in place.

8.2.5 Register File

The ECP has 22 registers, each of which holds a 163-bit value. Registers address are 5-bit long. Register numbers 1, 3, 6, and 20 are dedicated to store the input parameters x, b, y and k respectively. The register file supports 2 reads and 1 write.

8.3 Arithmetic Unit Architecture

The arithmetic unit consists of 4 structures – the multiplier, squarer, the basic ALU for add and shift operations and the zero detection unit.



SLL – Shift left

Fig 8.4: Arithmetic Unit

The zero detect unit verifies that the result from the ALU is not zero. The arithmetic unit gets the m-bit operands from the register file. Depending on the control signal, the appropriate operation is performed. The result obtained from the arithmetic unit is compared to ascertain that it is not zero and is written back to the registers or to the data memory.

8.3.1 Multiplier

The architecture of the multiplier is a digit serial/parallel architecture proposed by [HGK03]. This type of multiplier is versatile as it can work over a wide range of finite fields and is not restricted to a particular form of irreducible polynomial. The multiplier is scalable in the sense that the digit size can be selected as desired.

The fundamental idea behind the digit-serial architecture is that more than one bit at a time is processed. It gives a fair tradeoff between space and performance. A new method for degree reduction is also proposed by [HGK03].

The table below summarizes the time-area requirements for different multiplier architectures for the field GF (2^m).

Type	Area complexity	Time Complexity
Bit-Parallel	$O(m^2)$	O(1)
Bit-Serial	O(m)	O(m)
Digit-serial	O(mD)	O(m/D)

D – digit size.

Table 8.3: Time-area analysis for multiplier architectures [HGK03]

There are two approaches for digit-serial multiplication:

1. Least Significant Digit (LSD) First Architecture
2. Most Significant Digit (MSD) First Architecture

In the Least Significant Digit First mode, the multiplication takes place from right-to-left, whereas, it is the opposite in the Most Significant Digit First mode.

[SP96] compares the hardware requirements, latency and critical-path length of both the above approaches. The partial product obtained in the LSD approach is m+d-2 bits while that obtained by MSD is m-1 bits long for the field GF (2^m) (d is the digit size), so less number of bits have to be stored in the MSD approach. Also the latency in the MSD approach is [m/d] clock cycles whereas it is [m/d] +1 cycles in LSD first architecture.

The implementation given here uses the MSD multiplier architecture.

Let A and B be the two polynomials to be multiplied.

$$A = \sum_{i=0}^{m-1} a_i x^i$$

$$B = \sum_{i=0}^{m-1} b_i x^i$$

Now, in the bit serial approach, one bit of B will be multiplied with A in each clock cycle

$C = A * B = A(x) . B(x) \bmod P(x)$

where P(x) is the irreducible polynomial of degree m.

$C = [b_0 A(x) + b_1 x A(x) + ...+ b_{m-1} x_{m-1} A(x)] \bmod P(x).$

The above equation can be rewritten as follows by grouping multiplies of x and factoring out x

$C(x) = [... [[A(x) b_{m-1}] x \bmod P(x) + A(x) b_{m-2}] x \bmod P(x) + ...+ A(x) b_1] x \bmod P(x) + A(x) b_0$

In the digit-serial approach, instead of multiplying one bit of B at a time, d-bits of B are inputted (d – digit size), thereby reducing the number of clock cycles needed to m/d.

The digit size d can be selected as desired.

B can be represented as follows

$$B = \sum_{j=0}^{d-1} b_{i\cdot d+j} x^j \quad \text{for } 0 \le i \le m-2$$

$$(m \bmod d)-1$$

$$B = \sum_{i=0} b_{i.d+j}\, x^j \quad \text{for } i = m-1$$

Using this representation, the algorithm for MSD first multiplication is as shown below;

C (0) = 0

For k in 1 to m/d loop

Z (k) = A [m-1 downto 0] * B [((m/d-k)*d+ (d-1)) downto ((m/k-k)*d)]

C (k) = C (k-1) + shift_left (Z (k), d)

end loop.

In each iteration, A is multiplied with d bits of B, resulting in an m+d-1 bit polynomial. This has to be reduced to degree m-1. So, instead of carrying out the reduction process at the end, each computation of Z (k) is reduced to degree m-1. Therefore higher degree polynomials do not have to be stored.

Degree Reduction

To reduce Z (k) modulo P (x), a new and efficient method for degree reduction proposed by [HGK03] is used.

A polynomial y(x) of degree d-1 is used so that

Z (x) + P (x) Y (x) = T (x) = Z (x) mod P (x)

The computation of the coefficients of Y (x) is fairly easy and requires only AND and XOR gates as demonstrated by the following example.

Example: The computation of Y (x) for d = 4 is as shown

For z(x) + p(x) y(x) = z(x) mod p(x) , the d most significant bits of p(x)y(x) and z(x) must be identical. This fact is used to compute the coefficients of y.

$y_3 = z_{11}$

$y_2 = z_{10} + p_7 . z_{11}$

$y_1 = z_9 + p_7 . z_{10} + p_7 . z_{11} + p_6 . z_{11}$

$y_0 = z_8 + p_7 . z_{10} + p_7 . z_{11} + p_6 . z_{10} + p_5 . z_{11}$

Also when trinomials or pentanomials are used as the irreducible polynomial in, most of the coefficients of P (x) are zero, therefore most of the coefficients of Y (x) become coefficients of Z (x), which is a simple assignment.

8.3.2 Squarer Architecture

The squarer architecture used in this work is a bit-parallel architecture proposed by [WU99] i.e., the square can be computed in one clock cycle. The architecture makes use of only XOR gates and so the net delay is considerably less. This squarer is designed for trinomials or pentanomials as the irreducible polynomial, which results in a smaller gate count and small critical path length than using other arbitrary irreducible polynomials.

When the irreducible trinomial is used to generate the field GF (2^n), closed form expressions are developed using the coefficients of the input polynomial.
The irreducible trinomial is represented by the equation $f(x) = x^m + x^k + 1$ where $1 \leq k \leq m/2$.

The squaring algorithm uses three cases for $k = 1$, $1 < k < m/2$ and $k = m/2$.
If the input polynomial is $A = \sum a_i x^i$ for i between 0 and m-1
The square of A is given by
$A^2 = \sum a_i x^{2i}$ for i between 0 and m-1

$= \sum a_i x^i$ for i between 0 and 2m-2
and
$a_i = a_{i/2}$ if i is even
$= 0$ otherwise

Example

If $f(x) = x^5 + x^2 + 1$

Let A be the input polynomial $A = \sum a_i x^i$ and $C = \sum c_i x^i$ be the output polynomial, Then

$c_0 = a_0 + a_1$

$c_1 = a_4$

$c_2 = a_2$

$c_3 = a_0 + a_4$

$c_4 = a_3$

:

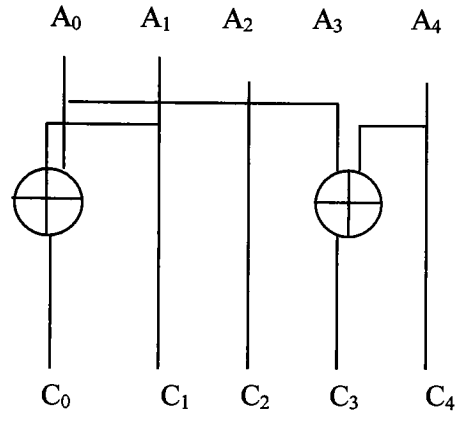


Fig 8.5: Example of Squarer

The number of XOR gates required for the squarer is (m-1)/2 and time delay is 2 times the delay of an XOR gate.

8.3.3 Inversion Architecture

The inverse is computed using Fermat's Little theorem. The inverse operation is performed only once when converting projective coordinates back to affine coordinates. The equation for computing inverse is

$A^{-1} = A^{2^{m}-2}$

The exponentiation is performed using the left-to-right binary exponentiation. The squarings and multiplications required are performed using the squarer and multiplier. Thus, the inversion operation is basically a combination of multiplications and squarings.

Pseudo Code for the inversion operation
INPUT: A, m
OUTPUT $A^{-1} = A^{2^{m}-2}$

Calculate $k = 2^m - 2$, $k_i \in 0, 1$
Initialize $t = 1$
for i in m-1 downto 0 **loop**
    $t = t * t$
    if $k_i = 1$, then
        $t = t * A$
    end if
end loop
Return $t = A^{-1}$

Example:
To calculate A^{165}
165 = 10100101

i	7	6	5	4	3	2	1	0
k_i	1	0	1	0	0	1	0	1
t	A	A^2	A^5	A^{10}	A^{20}	A^{41}	A^{82}	A^{165}

The binary representation of $k = 2^m - 2$ is scanned starting from the most significant bit. In each iteration a variable t is multiplied to itself. If the scanned bit is a 1, then t is multiplied with A.

Chapter 9

9.1 Results and Performance Analysis

This chapter discusses the results obtained from the implementation of the elliptic curve processor on an FPGA. Timing and area results are obtained for all the major modules in the design and for the entire system.

Two Target FPGAs were used for the implementation – Altera Flex10K70 and Xilinx Virtex XCV1000. The Xilinx FPGA has more number of gates, dual ported memory and incorporates fast carry logic. It also has 404 input-output pins enabling higher order field designs to be successfully tested. The Altera FPGA has dual port memory, lesser number of gates and was used to compare the performance of the elliptic curve processor on two different platforms.

The elliptic curve processor was tested for different field sizes. First the results obtained from the combinatorial structures are presented and then for the scalar multiplication process. For the combinatorial elements the development tools provided information about the maximum path delay, which describes the longest path in the design. Area results indicate the utilization of the combinatorial elements in the FPGA (i.e., CLB or LAB), which is a standard way of defining area utilization of FPGA implementations. Due to very long compile and place and route times and limited system resources, some of the area results for the higher order fields had to be estimated through linear approximation, since area increased linearly with the field size. Timing results were obtained from Synopsys synthesis for designs with very long compilation time, which were fairly close to those obtained from implementation on the FPGA device.

9.1.1 Addition

Addition in field GF (2^n) is simply an XOR operation and so it is bit-parallel with O (1). The results obtained are tabulated below.

ms – milliseconds

ns - nanoseconds

Using Flex10K70

Field Size n	Area (LAB)	Path (ns)
8	8	25.6
113	113	25.6
155	155	25.6
163	163	25.6

Table 9.1a: Addition results in FLEX 10K70

Using XCV1000 –6

Field Size n	Area Logic (CLB)	Path (ns)
8	8	8.198
113	113	8.198
155	155	8.198
163	163	8.198

Table 9.1b: Addition results in XCV1000

Since addition is bitwise XOR, the timing results are similar for any size of the field. Moreover, the gate utilization increases linearly with increase in field size. The area results are the utilization of the logic blocks in the FPGA, which implement the gates i.e., the LAB in Flex and CLB in the Virtex device. Also the results obtained on the Virtex were far better than on the Flex device.

9.1.2 Multiplication

Multiplication is an expensive operation in terms of resources needed. Different digit sizes were used to show how the multiplier architecture scales for the Xilinx device. Results are obtained for digit sizes 4, 16, 32.

Using Flex10K70 – Digit Size – 4 bits

Field Size n	Area (LAB)	Path (ns)
8	77	31.0
113	Design does not fit into device.	
155		
163		

Table 9.2a: Multiplication results in Flex10K70

For larger field sizes, the multiplier does not fit into the Altera device as it is a small device in terms of LABs present on the chip.

Using XCV1000 -6 - Digit Size – 4 bits

Field Size n	Area (CLB)	Path (ns)
8	63	16.575
113	889	325.848
155	1220	446.952
163	1283	470.028

Table 9.2.1b: Multiplication results in XCV1000 with d = 4

Using XCV1000 -6 - Digit Size – 16 bits

Field Size n	Area (CLB)	Path (ns)
113	1298	245.040
155	1780	336.11
163	1872	353.464

Table 9.2.2b: Multiplication results in XCV1000 with d = 16

Using XCV1000 -6 - Digit Size – 32 bits

Field Size n	Area (CLB)	Path (ns)
113	1837	226.342
155	2519	310.469
163	2649	326.493

Table 9.2.3b: Multiplication results in XCV1000 with d = 32

The timing results for 16-bit and 32-bit digit are almost similar, indicating that the speed does not increase linearly with the digit size. When the digit size is higher, the number of multiplications to be performed is lesser and so the multiplication processing time decreases [Sec. 8.3.1]. However there is an increase in the number of squarings and addition operations required, which relatively increases the processing time. There is an appreciable speedup as digit size increases from 8 to 16. However this speedup is relatively constant from d = 16 to d = 32. Area increases with digit size. A good choice for the digit size for cryptographic implementations would be between 4 and 16.

9.1.3 Inversion

This is the most time and resource consuming operation.

Using Flex10K70

Field Size n	Area (LAB)	Path(ns)
8	87	238.1
113	Design does not fit in device.	
155		
163		

Table 9.3a: Inversion results in Flex10K

Using XCV1000

Field Size n	Area (CLB)	Path (ns)
8	42	102.733
113	593	1451
155	813	1990
163	855	2093

Table 9.3b: Inversion results in XVC1000

The time taken for an inversion operation, obtained from synthesis on the target FPGA, for the field GF (2^{163}) is about 2.1 ms indicating that inversion is a slow operation. For the field GF (2^8), an inversion of element A would be calculated as $A^{-1} = A^{2^{m-2}} = A^{254}$. 254 is represented in binary as $(11111110)_2$, so the inversion operation requires 8 squarings and 6 multiplications (Sec 6.5.1). For higher order fields, the number of 1s in

the binary representation of 2^m-2 is higher, increasing the number of operations. 2^163-2 is a very large number of the order of 10^{49}, and inversion in this field requires $\log(10^{49})$ squarings and $\frac{1}{2}(\log(10^{49}) +1)$ multiplications. Since inversion is required only in a last step in the Montgomery multiplication algorithm when projective coordinates are converted back into affine, this result does not affect the overall scalar multiplication operation. However if affine coordinates where used in the implementation, where inversion operation is needed in every point double and point addition, the scalar multiplication would indeed be quite slow.

9.1.4 Squaring

Using Flex10K70

Field Size n	Area (LAB)	Path (ns)
8	8	12.8
113	113	15.6
155	155	15.6
163	163	15.6

Table 9.4a: Squaring results in Flex10K

Using XCV1000Field Size n	Area (CLB)	Path (ns)
8	8	8.495
113	113	8.495
155	155	8.495
163	163	8.495

Table 9.4b: Squaring results in XCV1000

The squarer implemented is a bit parallel squarer made up of XOR gates. The square can be computed in 8.5 ns on the Virtex and on 15.6 ns on the Flex.

9.1.5 Scalar Multiplication Process

This is the most important operation in elliptic curve cryptosystems. The scalar multiplication process is a series of point additions and point doublings, which in turn are

composed of field addition, inversion and multiplication operations. Inversion is eliminated by using projective coordinates for the representation of points on the elliptic curve.

Using Flex10K70

Field Size n	Area (LAB)	Speed (ns)
8	212	64.9
113	Does not fit into specified device	
155		
163		

Table 9.5a: Scalar Multiplication results in Flex10K

Using XCV1000

Field Size n	Area (CLB)	Speed (ns)
8	155	57.376 (17.429MHz)
113	2192	810.436
155	3006	1111
163	3440	1169

Table 9.5b: Scalar Multiplication results in XCV1000

For larger field sizes the design does not fit into the Flex device as it has only 468 LABs. **Device Utilization** is 56% for 163-bit scalar multiplication on the Virtex XCV1000. The entire scalar multiplication can be computed in 1.16 ms.

Power Consumption

The power consumption measured for the Virtex device is 32.57MW. The power consumption was computed using the XPower tool available with the Xilinx ISE. XPower estimates the power the design consumes.

9.2 Comparison with other implementations

The elliptic curve processor developed in this work performs a scalar multiplication operation in 1.16 milliseconds for GF (2^{163}). The following table compares the performance with other leading implementations of elliptic curve cryptosystems.

Implementation	Fields	Platform	Time
[AMV93]	GF (2^{155})	VLSI 40 MHz	3.9ms est.
[SES98]	GF (2^{155})	VLSI 40 MHz	18.4 ms
[Ros98]	GF ($((2^4)^2)^{21}$	Xilinx FPGA XC4062, 16MHz	4.5 ms
[LMW00]	GF (2^{113})	Xilinx FPGA XCV300, 45MHz	3.7 ms
[OP00a]	GF (2^{167})	Xilinx FPGA XCV400E, 76.7MHz	0.21 ms
[OTI00]	GF(2^{163})	Altera EPF10K250	45 ms
[GBK01]	GF ($2^{128} - 2^{97} - 1$)	TI MSP430x33x, 1 MHz	3.4sec
[AYK00]	GF (p)	32-bit ARM, 80MHz	44.8 ms
[WBP00]	GF ($(2^8 - 17)^{17}$)	Intel 8051	8.37sec

Table 9.6: Comparison with past implementations

Chapter 10

10.1 Summary

This thesis presents a 163-bit implementation of an elliptic curve processor for GF (2^n) on reconfigurable hardware. The processor is capable of performing the main operation of elliptic curve cryptosystems, which is the elliptic curve scalar multiplication. This operation can be performed by the processor in 1.16msec which is considerably faster than most of the other documented implementations. Since for fields greater than 100, the place and route caused the system to crash, the timing results for these fields were obtained from synthesizing the design using Synopsys. In a practical setting, better tools and resources would be used to place the design on the target device, leading to better results. Another implementation [OTI00] on the Altera FPGA device is slower than implementations on Xilinx as demonstrated from the results in this thesis. The design by [LUT03] is the fastest reported implementation of ECC, which uses the Look up Tables for multiplication and squarings, leading to extremely fast calculations. Also the implementation is optimized for specific fields and Koblitz curves.

The architecture used for the processor is the popular MIPS architecture, which performs finite field arithmetic as opposed to the normal arithmetic. So the elliptic curve processor is basically a general-purpose processor with a specialized function. The design of the processor is highly flexible; it can be reprogrammed easily by changing the instructions in the instruction memory. Also new instructions can be incorporated as needed.

10.2 Recommendation for further work

The elliptic curve parameters have to be loaded into the registers before the start of processing. It is a good idea to implement generation of random curves and points on the chip itself, which will make it more practical.

The run time reconfiguration feature of FPGAs can be exploited for future research which would dynamically reconfigure the FPGA with different algorithms, functional units or a different cryptographic processor altogether.

Another consideration would be to have multiple functional units to further increase the speed of computations since most of the algorithms can be parallelized, keeping in mind the area requirements especially when developed for constrained environments.

Bibliography

[ABV89] D. Ash, I. Blake and S. Vanstone, "Low complexity normal bases", 1989.

[ALTERA] Altera Website, <www.altera.com>

[AMV93] G. B. Agnew, R. C. Mullin, S. A. Vanstone, "An implementation of elliptic curve cryptosystems over F_2155", in *IEEE Transactions on Selected Areas in Communications*, 1993.

[AYK00] M. Aydos, T. Yanik and C. K. Koc, "A High-Speed ECC-based Wireless Authentication Protocol on an ARM Microprocessor", in *16th Annual Computer Security Applications Conference*, December 2000.

[BDM01] M. Brown, Darrel Hankerson, Julio Lopez and Alfred Menezes, "Software implementation of the NIST Elliptic Curve over Prime Fields", in *Proceedings of CT-RSA,* 2000.

[BDG02] M. Bednara, M. Daldrup, J. von zur Gathen, J. Shokrollahi, J. Teich, "Reconfigurable Implementation of Elliptic Curve Crypto Algorithms", in *International Parallel and Distributed Processing Symposium, IPDPS*, April 2002.

[BDT02] M. Bednara, M. Daldrup, J. Teich, J. von zur Gathen, J. Shokrollahi, "Tradeoff Analysis of FPGA based elliptic curve cryptography ", in *IEEE* 2002.

[CERT98] Certicom White Paper, "The Elliptic Curve Cryptosystem for Smart Cards", May 1998.

[CERT00] Certicom White Paper, "Elliptic Curve Cryptosystems", 2000

[CLH00] Jung Hee Cheon, Dong Hoon Lee, Sang Geun Hahn and Seongtaek Chee, "Elliptic Curve Discrete Logarithms and Wieferich Primes", *in Proceedings of JW-ISC2000, pp. 25-26*, 2000.

[COP84] D. Coppersmith, "Fast evaluation of logarithms in field of characteristic two", *IEEE Trans, Inform Theory*, 1984.

[DAH97] D. Dahm, "Personal communication and *sci.crypt* postings", September 1997.

[DBV96] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem and J. Vandewalle, "A fast software implementation for arithmetic operation in GF (2^n)", in *Asiscrypt* 1996.

[DH76] W. Diffie and M. E. Hellman, "New directions in cryptography", in *IEEE Transactions on Information Theory*, 1976.

[DLA00] Darrel Hankerson, Julio Lopez Hernandez and Alfred Menezes, "Software Implementation of Elliptic Curve Cryptography over Binary Field", in *Proceedings of CHES* 2000.

[DMV01] Janaka Deepakumara, Howard M. Heys and R. Venkatesam, "FPGA Implementation of MD5 Hash Algorithm", in *Electrical and Computer Engineering Canandian Conference on, vol. 2, pp 919-924*, 2001.

[DW99] Andre DeHon and John Wawrzynek, "Embedded Tutorial: Reconfigurable Computing: What, Why and Implications for Design Automation", in *Design Automation Conference, page 610*, 1999.

[EJM02] M. Ernst, M. Jung, F. Madlener, S. Huss and R. Blumel, "A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over GF (2^n)." in *CHES* 2002.

[ENG99] Andread Enge, "Elliptic Curves and their applications to cryptography: An introduction", *Kluwer academic publishers*, September 1999.

[EYC00] A. J. Elbirt , W. Yip, B Chetwynd and C. Paar, "An FPGA Implementation and performance evaluation of the AES block cipher candidate algorithm finalists", in *The third AES Candidate conference*, April 2000.

[FEL89] David C. Feldmeier, "A High-Speed Software DES Implementation", *can be obtained by anonymous ftp from thumper.bellcore.com as /pub/crypt/des.ps.Z*, 1989.

[FL00] A. Murat Fiskiran and Ruby B. Lee, "Workload Characterization of Elliptic Curve Cryptography and other Network Security Algorithms for Constrained Environments", in *Proceedings of the 5th Annual IEEE International Workshop on Workload Characterization (WWC-5), pages 127 – 137*, November 2002.

[GBK01] J. Guajardo, R. Bluemel, U. Krieger and C. Paar, "Efficient Implementation of Elliptic Curve Cryptosystems on the TI MSP430x33x family of Microcontrollers" in *Proceedings of PKC* 2001.

[GHS00] P. Gaundry, F. Hess and N. P. Smart, "Constructive and destructive facets of Weil descent on elliptic curves." in *HP Labs Technical Report*, January 2000.

[GKP00] J. Großsch¨adl, G. A. Kamendje, E. Oswald, and R. Posch, "Elliptic Curve Cryptography in Practice", *in Graz University of Technology,* 2000.

[GP97] Jorge Guajardo and Christof Paar, "Efficient Algorithms for Elliptic Curve Cryptosystems", in *Advances in Cryptology*, 1997.

[GP02] J. Guajardo and C. Paar, "Itoh-Tsujii Inversion in Standard basis and its application in cryptography and code", in *Design, Codes and Cryptography, 25(2) : 207 – 216*, February 2002.

[GSS99] L. Gao, S. Shrivastava and G. Sobelman, "Elliptic curve scalar multiplier design using FPGAs" in *CHES* 1999.

[HAS01] M. A. Hasan, "Efficient Computation of Multiplicative Inverses for Cryptographic Applications", in *CACR Technical Report* 2001.

[HGK03] M. Hutter, J Groschadl and G. A Kamendje,"A versatile and scalable digit-serial/parallel multiplier architecture for finite fields GF (2^m)"; in *Information Technology: Coding and Computing*, 2003.

[HIT02] Yvonne Hitchcock, "The Security of Fixed versus Random Elliptic Curves", *in Information Security Research Centre, ISRC Workshop* 2002.

[HM04] D. Hankerson and A. Menezes, "Elliptic Curve Discrete Logarithm Problem", *in Auburn University and Waterloo University*, 2004.

[IT88] T. Itoh and S. Tsujii, "A Fast Algorithm for Computing Multiplicative Inverses in GF (2^m) using Normal Bases", in *Information and Computation, 78: 171-177*, 1988.

[IWD92] J. S. P. Ivey, S. Walker and S. Davidson, " An ultra-high speed public key encryption processor", in *Proceedings on the IEEE Custom Integrated Circuits Conference* 1992.

[KEN93] S. Kent, "RFC 1422: Privacy Enhancement for Internet Electronic Mail, Part 2: Certificate-Based key Management", *Internet Activities Board*, February 1993.

[KIN01] B. King, "An improved implementation of elliptic curves over GF (2^n) when using projective point arithmetic" in *SAC* 2001.

[KOB87] N. Koblitz, "Elliptic Curve Cryptosystems", in *Mathematics of Computation*, 1987.

[KMH01] Kunio Kobayashi, Hikaru Morita and Mitsuari Hakuta, "Multiple Scalar-Multiplication Algorithm over Elliptic Curve", in *IEICE Trans. Information and Systems, Vol. E84-D*, February 2001.

[KOB92] N. Koblitz, "CM-curves with good cryptographic properties", *Advances in Cryptology – CRYPTO* 1991, *Lecture Notes in Computer Science, volume 576, Springler-Verlag, pages 279-287*, 1992.

[LD98] J. Lopez and R. Dahab, "Improves Algorithms for elliptic curve arithmetic in GF (2^m)", in *Technical Report, IC-98-39*, October 1998.

[LD99] J. Lopez and R. Dahab, "Fast multiplication on elliptic curves over GF (2^m) without precomputation", in *CHES* 1999.

[LD00] J. Lopez and R, Dahab, "An Overview of Elliptic Curve Cryptography", in *Technical Report, Institute of Computing, State University of Campinas, Brazil, May 2000.*

[LHC01] Zhi Li, John Higgins, Mark Clement, "Performance of Finite Field Arithmetic in an Elliptic Curve Cryptosystem", in *Ninth International Symposium in Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, August 2001.

[LKP03] Mun-Kyu Leeo, Jin Wook Kim and Kunsoo Park, "Security of using Special integers in elliptic scalar multiplication", *in 14th Australasion Workshop on Combinatorial Algorithms*, July 2003.

[LL02] P. H. W. Leong and I. K. H. Leung," A microcoded elliptic curve processor using FPGA technology", in *VLSI Systems, IEEE*, October 2002.

[LMW00] K. H. Leung, K. W. Ma, W. K. Wong, P. H. W. Leong, "FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor", in *IEEE* 2000.

[LM95] R. Lercier and F. Morain, "Couting the Number of points on Elliptic Curves over Finite Fields: Strategies and Performances", in *EUROCRYPT* 1995.

[LL02] P. H. W. Leong and I. K. H. Leung," A microcoded elliptic curve processor using FPGA technology", in *VLSI Systems, IEEE*, October 2002.

[LN94] R. Lidl and H. Niederreiter, "Introduction to finite fields and their applications", in *Cambridge University Press, Cambridge,* UK, 1994.

[LUT03] Jonathan Lutz, "High performance elliptic curve cryptographic co-processor", in *Masters Thesis, University of Waterloo,* 2003.

[MAR01] Alan Marshall, "Reconfigurable approach supersedes VLIW/ superscalar", *in EE Times*, 2001

[MAU94] U. Maurer, "Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete logarithms", *Advances in Cryptology – CRYPT '94, Springer-Verlag*, 1994.

[MEN95] A. Menezes, "Elliptic Curve Cryptosystems. Cryptobytes", *Vol. 1 No. 2,* Summer 1995.

[MH93] Anthony M. Michel and Charles J. Herget, "Applied Algebra and Functional Analysis", in *Dover Publications, Inc, NY*, 1993.

[MIL85] V.S. Miller, "Use of Elliptic Curves in Cryptography", *in CRYPTO* 1985.

[MOV89] R.C. Mullin, I. M. Onyszchuk, S. A. Vanstone and R. M. Wilson, "Optimal Normal Bases GF(p^n)", *in Discrete Applied Mathematics, 22:149-161*, 1988/89.

[MOV96] A. Menezes, P. Van Oorschot and S. Vanstone, "Handbook of Applied Cryptography ", in *CRC Press* 1996.

[NGC03] Nghi Nguyen, Kris Gaj, David Caliga and Tarek El-Ghazawi, "Implementation of Elliptic Curve Cryptosystems on a Reconfigurable

Computer", in *IEEE Conference on Field Programmable Technology, FPT* 2003.

[OBP03] Siddika Berna Ors, Lejla Batina and Bart Preneel, "Hardware Implementation of Elliptic Curve Processor over GF (p)", in IEEE ASAP Conference, 2003.

[Odl84] A. M. Odlyzko, "Discrete logarithms in finite fields and their cryptographic significance", *Proc EUROCRYPT* 1984.

[OM98] J. H. Oh and S. J. Moon, "Modular Multiplication Method", in *Computers and Digital Techniques, IEEE Proceedings-, Volume: 145,* July 1998.

[OP00a] G. Orlando and C. Parr, "A High-Performance Reconfigurable Elliptic Curve Processor for GF (2^ n)", in *Workshop of Cryptographic Hardware and Embedded Systems* (CHES 2000).

[OP00b] G. Orlando and C. Parr, "Squaring Architecture for GF (2^m) and its Applications in Cryptographic Systems", in *Electronic Letters* June 2000.

[OP01] G. Orlando and C. Parr, "A Scalable GF (p) Elliptic Curve Processor Architecture for Programmable Hardware", in *CHES* 2001.

[OTI00] Souichi Okada, Naoya Torii, Kouichi Itoh, and Masahiko Takenaka, "Implementation of elliptic curve cryptographic coprocessor over GF (2^m) on an FPGA", in *Cryptographic Hardware and Embedded Systems (CHES), Springer-Verlag* 2000.

[OSW02] Elisabeth Oswald, "Introduction to Elliptic Curve Cryptography", in *Institute of Applied Information Processing and Communication, Austria,* July 2002.

[PAR95] Christof Paar, "Some remarks on efficient inversion in finite fields", *in IEEE International Symposium on Information Theory,* September 1995.

[PAR99] C. Parr, "Implementation Options for Finite Field Arithmetic for Elliptic Curve Cryptosystems", in *ECC* 1999.

[PFS99] C. Parr, P. Fleischmann and P. Soria-Rodriguez, "Fast Arithmetic for public-key algorithms in Galois fields with composite exponents, in *IEEE Transactions on Computers,* October 1999.

[PIE00] Henna Pietilainen, "Elliptic Curve Cryptography on Smart Cards", in *Masters Thesis, Helsinki University of Technology*, 2000.

[PT01] Vikram Pasham and Steve Trimberger, "High-Speed DES and Triple DES Encryption/Decryption", in *Xilinx Application Note: Virtex-E Family and Virtex-II Series*, August 2001.

[ROS98] M. Rosner, "Elliptic Curve Cryptosystems on Reconfigurable Hardware", *Masters Thesis, Worcester Polytechnic Institute,* May 1998.

[RIV92a] R. L. Rivest, "Response to NIST's proposal", in *Communication of the ACM*, July 1992.

[ROS99] M. Rosing, "Implementing Elliptic Curve Cryptography", 1999.

[RSA00] RSA Laboratories, "RSA Laboratories' Frequently Asked Questions About Today's Cryptography, Version 4.1", *RSA Security Inc*, 2000.

[SAE96] M. Saeki, "Elliptic Curve Cryptosystems", *M. Sc. Thesis, McGill University of Computer Science,* 1996.

[SB03] Devarkal Siddaveerasharan and Duncan A. Buell, "Elliptic Curve Arithmetic on Reconfigurable Hardware", in *MAPLD International Conference*, 2003.

[SES98] S. Sutikno, R. Effendi and A. Surya, "Design and Implementation of arithmetic processor F_2^{155} for elliptic curve cryptosystems" in *the 1998 IEEE Asia-Pacific Conference on Circuits and Systems, pages 647-650*, 1998.

[SHA97] M. Shand, "A Case Study in Algorithm Implementation in Reconfigurable Hardware and Software", *in Field Programmable Logic and Applications*, London, 1997.

[SIM92] G. J. Simmons, "Contemporary Cryptology – The Science of Information Integrity , in *IEEE Press*, 1992.

[SMA00] N.P. Smart, "A comparison of different finite fields for use in elliptic curve cryptosystems", June 2000.

[S0095] R. Schroeppel, H. Orman, S. O'Mally and O. Spatscheck, "Fast key Exchange with elliptic curve systems", in *CRYPTO* 1995.

[SS01] Y. Sakai and K. Sakurai,"Efficient Scalar Multiplications on Elliptic Curves with Direct Computation of Several Doublings", in *IEICE Trans. Fundamentals*, 2001.

[ST03] Akashi Satoh and Kohji Takano, "A scalable dual-field elliptic curve cryptographic processor", *in IEEE Transactions on Computers, pg 449*, April 2003.

[USG00] U.S. Department of Commerce/National Institute of Standards and Technology: Digital Signature Standard (DSS), Federal Information Processing Standards Publication FIPS PUB 186-2, January 2000.

[WAL91] Bob Walder, "Key Escrow vs Key Recovery", *A NSS Group White Paper*, 1991.

[WM03] R. W. Ward and Dr. T. C. Molteno, "Efficient Hardware Calculation of Inverses in GF (2^8)", *in Proceedings of ENZCon'03, University of Waikato, NZ*, September 2003.

[WBP00] A. Woodbury, D. Bailey and C. Paar, "Elliptic Curve Cryptography on Smart Cards without Coprocessors", in *Fourth Smart Card Research and Advanced Applications,* September 2000.

[WPC01] A. Weimerskirch, C. Paar and Shantz S. Chang,"Elliptic Curve Cryptography on a Palm OS Device", in *ACISP* 2001.

[WSC89] R. Waxman, L. Saunders and H. Carter, "VHDL links design, test and maintenance", in *IEEE Spectrum*, May 1989.

[WU99] H. Wu,"Low complexity bit-parallel finite field arithmetic using polynomial basis", in CHES 1999.

[XILINX] Xilinx Website, www.xilinx.com.

[YOR92] E. V. York, "Elliptic curves over finite fields", *Master's Thesis, George Mason University*, May 1992.

Appendix A

Matlab Code for Scalar Multiplication

```
%*************************************************************************
% File - madd.m
% This is the montgomery add function used for point addition
% Author - Aarti Malik
% Last Updated - 03/18/04
%
*************************************************************************


function [X11,Z11]= madd (X1,Z1,X2,Z2,x)

X11 = (( X1 * Z2) * (X2 * Z1 )) + x * (((X1 * Z2) + (X2 * Z1)) * ((X1 *
Z2) + (X2 * Z1)));
Z11 = ((X1 * Z2) + (X2 * Z1)) * ((X1 * Z2) + (X2 * Z1));



*************************************************************************
% File - mdouble.m
% This is the montgomery double function used for point doubling
% Author - Aarti Malik
% Last Updated - 03/18/04
%
*************************************************************************

function [X1,Z1] = mdouble(X,Z,b)
X1 = (X * X * X  * X) + b * (Z * Z * Z * Z);
Z1 = X * X * Z * Z;
```

```
%*************************************************************************
% File - projtoaffine.m
% This function converts projective coordinates back to affine.
% Author - Aarti Malik
% Last Updated - 03/19/04
%
*************************************************************************

function [xk,yk] = projtoaffine(X1,Z1,X2,Z2,x,y)
xk = X1 * inverse(Z1);
yk = ((xk + x) * (X2* inverse (Z2) + x) + x*x + y) * ((xk + x)*
inverse(x) + y);




*************************************************************************
% File - scalarmult.m
% This is the montgomery scalar multiplication algorithm for GF(2^n)
% Author - Aarti Malik
% Last Updated - 03/25/04
%
*************************************************************************
clear all;

x = gf(165,8,265); % Galois vector in GF(2^3)
y = gf(90,8,265);
k = gf([0 1 0 1 1 0 1 0], 8, 265);
b = gf(43, 8, 265);

% Converting affine to projective coordinates.
X1=x;
Z1=gf(1,8,265);
x2=x*x;
x4=x2*x2;
X2=x4 + b;
Z2=x2;


for l=2:8
    if (k(l) == 1)
        [X1,Z1] = madd(X1,Z1,X2,Z2,x);
        [X2,Z2] = mdouble(X2,Z2,b);
    else
        [X2,Z2] = madd(X2,Z2,X1,Z1,x);
        [X1,Z1] = mdouble(X1,Z1,b);
    end

end


[xk,yk] = projtoaffine(X1,Z1,X2,Z2,x,y);
```

Appendix B

Scalar Multiplication Instruction in Memory

```
Depth = 256;
Width = 32;
Address_radix = HEX;
Data_radix = HEX;

Content
Begin

[00..FF] : 00000000;


00 : 8C0A0006; -- LOAD $10, 06 --LOAD 10, 0
01 : 00000000; -- LOAD $3, 01 -- LOAD 3, Y
02 : 8C040002; -- LOAD $4, 02 -- LOAD 4, 1 -- Z1
03 : 82850000; -- SQR $5, $20 -- X^2         -- Z2
04 : 8C090005; -- LOAD $9, 05 -- LOAD 8, x40
05 : 00000000;  --8C140000; -- Load $20, 00 -- load 20, x
06 : 028A1020; -- xor $2, $20, $10  --load 2, x --X1
07 : 00000000; -- LOAD $6, 03 -- LOAD 6, B
08 : 00000000; -- LOAD $1, 04 -- LOAD 1, K
09 : 80A70000; -- SQR $7, $5 -- X^4
0A : 00E64020; -- XOR $8, $7, $6 -- X^4 + B -- X2
0B : 0029A804; -- AND $21, $1, $9 -- AND n, k
0C : 01204801; -- SLR $9, $9


0D : 112A0021;  -- BEQ $9, $10  ,2E
0E : 00000000;
0F : 11550010; -- BEQ $21, $10, 1E
10 : 00000000;

-- WHEN K(i) = 1

11 : 20455800; -- MUL $11, $2, $5 -- MUL X1, Z2
12 : 21046000; -- MUL $12, $8, $4 -- MUL X2, Z1
13 : 810D0000; -- SQR $13, $8 -- SQR X2
14 : 80AE0000; -- SQR $14, $5 -- SQR Z2
15 : 216C7800; -- MUL $15, $11, $12 -- X1Z2X2Z1
16 : 016C8020; -- XOR $16, $11, $12 -- X1Z2 + X2Z1
17 : 21AE2800; -- MUL $5, $13, $14 -- Z2 = X2^2 Z2^2
18 : 81B10000; -- SQR $17, $13 -- X^4
19 : 81CE0000; -- SQR $14, $14 -- Z^4
1A : 20CE9000; -- MUL $18, $6, $14 -- bZ^4
1B : 82040000; -- SQR $4, $16 -- sqr(X1Z2 + X2Z1)
1C : 02324020; -- XOR $8, $17, $18 -- X^4 + bZ^4
1D : 20949800; -- MUL $19, $20, $4 --  sqr(X1Z2 + X2Z1) * x
1E : 026F1020; -- XOR $2, $19, $15 --sqr(X1Z2 + X2Z1) * x + X1Z2X2Z1
1F : 100000EB;

-- WHEN K(i) = 0
```

```
20 : 20455800; -- MUL $11, $2, $5 -- MUL X1, Z2
21 : 21046000; -- MUL $12, $8, $4 -- MUL X2, Z1
22 : 804D0000; -- SQR $13, $2 -- SQR X1
23 : 808E0000; -- SQR $14, $4 -- SQR Z1
24 : 216C7800; -- MUL $15, $11, $12 -- X1Z2X2Z1
25 : 016C8020; -- XOR $16, $11, $12 -- X1Z2 + X2Z1
26 : 21AE2000; -- MUL $4, $13, $14 -- Z1 = X1^2 Z1^2
27 : 81B10000; -- SQR $17, $13 -- X^4
28 : 81CE0000; -- SQR $14, $14 -- Z^4
29 : 20CE9000; -- MUL $18, $6, $14 -- bZ^4
2A : 82050000; -- SQR $5, $16 -- Z2 = sqr(X1Z2 + X2Z1)
2B : 02321020; -- XOR $2, $17, $18 -- X^4 + bZ^4
2C : 22859800; -- MUL $19, $20, $5 -- sqr(X1Z2 + X2Z1) * x
2D : 01F34020; -- XOR $8, $15, $19 -- Z2 = sqr(X1Z2 + X2Z1) * x +
X1Z2X2Z1
2E : 100000DC;

-- PROJECTIVE TO AFFINE

2F : 408B0000; -- INV $11, $4 -- 1/Z1
30 : 40AC0000; -- INV $12, $5 -- 1/Z2
31 : 828D0000; -- SQR $13, $20 -- x^2
32 : 21627000; -- MUL $14, $11, $2 -- X1/Z1 - xK
33 : 210C7800; -- MUL $15, $12, $8 -- X2/Z2
34 : 006D8020; -- XOR $16, $13, $3 -- x^2 + y
35 : 01D45820; -- XOR $11, $14, $20 -- X1/Z1 + x
36 : 01F46020; -- XOR $12, $15, $20 -- X2/Z2 + x
37 : 42910000; -- INV $17, $20 --1/x
38 : 218B9000; -- MUL $18, $12, $11 -- (X1/Z1 + x)(X2/Z2 + x)
39 : 21719800; -- MUL $19, $11, $17 -- (X1/Z1 + x)/x
3A : 02505820; -- XOR $11, $18, $16 -- (X1/Z1 + x)(X2/Z2 + x) + x^2 + y
3B : 02636020; -- XOR $12, $19, $3 -- (X1/Z1 + x)/x + y
3C : 216C6800; -- MUL $13, $11, $12 -- yK -- storing this in 24

3D : AC0E0007; -- STORE $14, 07 - x
3E : AC0D0008; -- STORE $13, 08 - y

3F : 8C190008;
40 : 8C180007;

End;
```